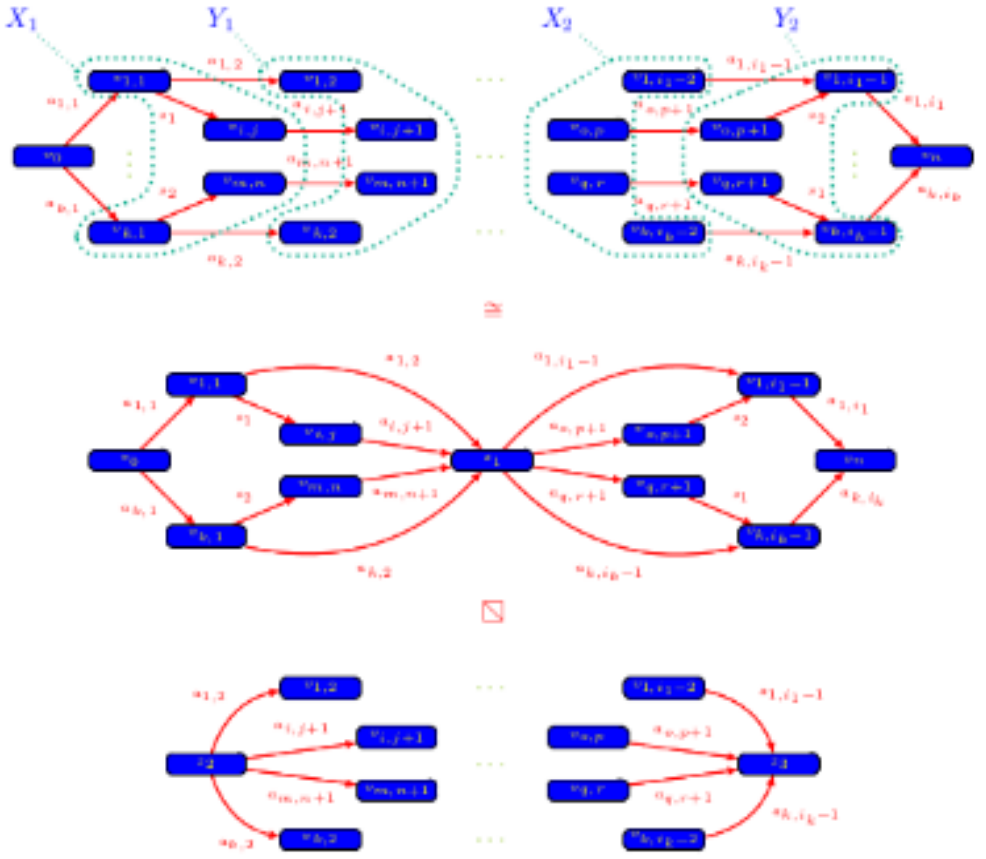
On the Automation of Periodic Hard Real-Time Processes

A Graph-Theoretical Approach



A.H. Boode

On the Automation of Periodic Hard Real-Time Processes

A Graph-Theoretical Approach

A.H. Boode

Faculty of Electrical Engineering, Mathematics and Computer Science, University of Twente

Robotics and Mechatronics Group

IDS Ph.D-Thesis Series No. 18-466 ISSN 2589-4730
Centre for Telematics and Information Technology
P.O. Box 217, 7500 AE Enschede, The Netherlands

This research was funded by the InHolland University of Applied Science

Title: On the Automation of Periodic Hard Real-Time Processes, A Graph-Theoretical Approach

Author: A.H. Boode

ISBN: 978-90-365-4551-8

ISSN: ISSN 2589-4730; IDS Ph.D-Thesis Series No. 18-466

DOI: 10.3990/1.9789036545518

Printed by Gildeprint - the Netherlands

ON THE AUTOMATION OF PERIODIC HARD REAL-TIME PROCESSES

A GRAPH-THEORETICAL APPROACH

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. T.T.M. Palstra,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 6 juni 2018 om 16.45 uur

door

Antoon Hendrik Boode
geboren op 13 April 1954
te 's-Gravenhage

Dit proefschrift is goedgekeurd door:

dr.ir. J.F. Broenink, promotor
prof.dr.ir. H.J. Broersma, promotor

Graduation committee

Chairman and Secretary	
prof.dr. J.N. Kok	University of Twente
Supervisors	
dr.ir. J.F. Broenink	University of Twente
prof.dr.ir. H.J. Broersma	University of Twente
Members	
prof.dr.ir. H.P.J. Bruyninckx	KU Leuven
dr.ir. C.J.M. Heemskerk	InHolland University of Applied Science
prof.dr. N. Litvak	University of Twente
prof.dr.ir. G.J.M. Smit	University of Twente
prof.dr. B. Vinter	University of Copenhagen

Summary

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. Reducing the number of context switches decreases the execution time and thereby increases the performance of the application.

Furthermore, the end-to-end processing time suffers from the idle time of the processor, because, for example, processes have to wait for controllers executing some task. By relaxing the rules for synchronous communication via channels in the process-algebraic specification language Communicating Sequential Processes (CSP), we are able to reduce the end-to-end processing time.

As we consider robotic applications only, often consisting of processes with identical periods, release times and deadlines, we restrict these applications to periodic real-time processes executing on a single-core mono-processor.

Because these processes can be represented by finite, deterministic, labelled, acyclic, directed multigraphs, we address these two problems using graph theory. We introduce a model of computation that, based on these graphs, shows an improved performance when we multiply these graphs. This multiplication is based on a synchronised graph product for which we have developed three versions; the Vertex-Removing Synchronised Product (VRSP), the Dot Vertex-Removing Synchronised Product (DVRSP) and the Extended Dot Vertex-Removing Synchronised Product (EVRSP). The VRSP is solely developed to reduce the number of context switches. The DVRSP and the EVRSP are an extension of the VRSP and deal with the reduction of the end-to-end execution time of a set of Periodic Hard Real-Time Control Processes (PHRCPs). Of course, these multiplications preserve the behaviour of the PHRCPs represented by these graphs.

Our research is based on three research questions, where we define the various graph products, prove that these products will give a performance gain (under certain conditions) and elaborate the numerical and combinatorial aspects of these graph products.

We introduce the notion of a consistent and an inconsistent set of graphs (representing periodic real-time processes). Consistency is based on the contraction of graphs together with the sink and the source of the Cartesian Product of these graphs, where the sink and the source have to be invariant over the graph multiplication by the synchronised product, VRSP. We show that consistency and associativity of the VRSP are closely related in the sense that a set of graphs under the VRSP is associative if all pairs of graphs in the set of graphs and their products under the VRSP are consistent.

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. But still, if the Periodic Hard Real-Time Control System (PHRCS) does not fulfil the requirements with respect to the deadline of its PHRCPs, calculating all possible products of two or more graphs may produce a set of graphs for which the processes they represent comprise a PHRCS that *will* fulfil the requirements with respect to deadline and memory occupancy.

To increase the chance that such a PHRCS exists, we develop two theorems that decompose the graphs into smaller graphs. Decomposition of the graphs gives a new set of graphs from which the VRSP gives outcomes that were not available in the original set of graphs. It could well be that these new outcomes contain a solution for the PHRCPs, whereas the original set of graphs did not contain a solution.

We show that the number of possible combinations of multiplications of graphs by the VRSP follows the Bell number (B_n) series if the multiplication is associative. Therefore we develop heuristics that calculate a set of multiplied graphs that may fulfil the requirements with respect to deadline and memory occupancy.

To emphasize the necessity of associativity, we study the multiplication by the VRSP when the multiplication is not associative. We give proof that this multiplication follows the Bessel number ($\tilde{B}_n$) series by calculating the number of different forests, where a set of multiplied graphs under the VRSP is represented by a binary tree. The numbers in the Bessel number series are a magnitude larger than the numbers in the Bell number series and this is, as for consistency, a reason why associativity is necessary.

All in all, we have five advantages provided by our graph theoretical approach:

- the length of the longest paths of the graphs is reduced, thereby reducing the number of context switches of the processes represented by these graphs,
- in a distributed computing system, for example, a processor-coprocessor combination, the end-to-end processing time of processes can be reduced.
- it eases the design by taking away the burden of separating the writing actions and reading actions in time, which eliminates the necessity of the modelling of a buffer,
- it gives more flexibility by indexing the reading actions,
- it allows multiple write actions to the same channel.

Samenvatting

In bepaalde single-core configuraties met één processor, b.v. embedded control systems zoals robotic applications die uit vele korte processen bestaan, kunnen de context switches van een proces een aanzienlijke hoeveelheid van de beschikbare processing power verbruiken. Het verminderen van het aantal context switches vermindert de executietijd en verhoogt daardoor de prestaties van de toepassing.

Bovendien is de end-to-end executietijd van de processen langer dan strict noodzakelijk, bijvoorbeeld omdat de processen moeten wachten op controllers die een taak uitvoeren. Door de regels voor synchrone communicatie via kanalen in de procesalgebraïsche specificatietaal Communicating Sequential Processes (CSP) te versoepelen, kunnen we de end-to-end executietijd verkorten.

Omdat we alleen rekening houden met robotic applications, vaak bestaande uit processen met identieke periodes, releasetijden en deadlines, beperken we deze applicaties tot periodieke real-time processen die worden uitgevoerd op een single-core mono-processor.

Omdat deze processen kunnen worden gerepresenteerd door finite, deterministic, labelled, acyclic, directed multigraphs, benaderen we deze twee problemen door middel van grafen theorie. We introduceren een verwerkingsmodel dat op basis van deze grafen verbeterde prestaties vertoont wanneer we deze grafieken vermenigvuldigen. Dit model is gebaseerd op een gesynchroniseerd graafproduct waarvoor we drie versies hebben ontwikkeld; het Vertex-Removing Synchronised Product (VRSP), het Dot Vertex-Removing Synchronised Product (DVRSP) en het Extended Dot Vertex-Removing Synchronised Product (EVRSP). Het VRSP is uitsluitend ontwikkeld om het aantal context switches te verminderen. Het DVRSP en het EVRSP zijn een uitbreiding van het VRSP en gaan over de reductie van de end-to-end executietijd van een verzameling Periodic Hard Real-Time Control Processes (PHRCPs). Natuurlijk behouden deze vermenigvuldigde grafen het gedrag van het PHRCPs vertegenwoordigd door deze grafen.

Ons onderzoek is gebaseerd op drie onderzoeksvragen, waarin we de verschillende graafproducten definiëren, bewijzen dat deze producten een prestatiewinst opleveren (onder bepaalde voorwaarden) en de numerieke en combinatorische aspecten van deze graafproducten uitwerken.

We introduceren het concept van een consistente- en een inconsistente reeks grafen (die periodieke real-time processen vertegenwoordigen). Consistentie is gebaseerd op de compositie van grafen samen met de sink en de source van het Cartesian product van deze grafen, waarbij de sink en de source invariant moeten zijn ten opzichte van de graafvermenigvuldiging door het gesynchroniseerde product, VRSP. We laten zien dat consistentie en associativiteit van het VRSP nauw verwant zijn in de zin dat een reeks grafen onder het VRSP associatief is als alle paren grafen in de reeks grafen en hun producten onder het VRSP consistent zijn.

Of een significante prestatiewinst al dan niet wordt behaald door het combineren van processen, hangt af van de verhouding tussen de context-switch tijd en de executietijd van de processen zelf; dit is duidelijk afhankelijk van het type hardware en besturingssysteem dat wordt gebruikt. Maar toch, als het Periodic Hard Real-Time Control System (PHRCS) niet voldoet aan de vereisten met betrekking tot de deadline van zijn PHRCPs, kan het berekenen van alle mogelijke producten van twee of meer grafen een reeks grafen opleveren waarvoor de processen die ze vertegenwoordigen, een PHRCS opleveren dat *zal* voldoen aan de vereisten met betrekking tot deadline en geheugenbezetting.

Om de kans te vergroten dat zo'n PHRCS bestaat, ontwikkelen we twee stellingen die de grafen ontleden in kleinere grafen. Decompositie van de grafen geeft een nieuwe reeks grafen waarvan het VRSP resultaten geeft die niet berekenbaar waren in de originele set grafen. Het zou best kunnen dat de producten van deze nieuwe reeks grafen een oplossing bevatten voor de PHRCPs, terwijl de originele set grafen geen oplossing bevatte.

We laten zien dat het aantal mogelijke combinaties van grafen door het VRSP de Bell number (B_n) reeks volgt als de vermenigvuldiging associatief is . Daarom ontwikkelen we heuristieken die een reeks vermenigvuldigde grafen berekenen die kunnen voldoen aan de vereisten met betrekking tot deadline en geheugenbezetting.

Om de noodzaak van associativiteit te benadrukken, bestuderen we de vermenigvuldiging met het VRSP wanneer de vermenigvuldiging niet associatief is. We bewijzen dat deze vermenigvuldiging de Bessel number ($\tilde{B}_n$) reeks volgt door het aantal verschillende forests te berekenen, waarbij een reeks vermenigvuldigde grafen onder het VRSP wordt vertegenwoordigd door een binary tree. De getallen in de Bessel number reeks groeien veel sneller en zijn van een andere orde dan de getallen in de Bell number reeks en dit is, wat de consistentie betreft, een reden waarom associativiteit noodzakelijk is.

Al met al zijn er vijf voordelen van onze grafentheoretische benadering:

- de lengte van de langste paden van de grafieken wordt verkleind, waardoor het aantal context switches van de processen die door deze grafen worden gerepresenteerd wordt verminderd,
- in een gedistribueerd computersysteem, bijvoorbeeld een processor / coprocessorcombinatie, kan de end-to-end executietijd van de processen worden verminderd.
- het vergemakkelijkt het ontwerpen van een PHRCS door het asynchroon maken van de schrijfacties en leesacties naar een kanaal in de tijd, waardoor de noodzaak van het modelleren van een buffer wordt geëlimineerd,
- het geeft meer flexibiliteit door de leesacties te indexeren,
- het staat meerdere schrijfacties op hetzelfde kanaal toe.

Contents

1

Introduction

In this thesis, we introduce and elaborate new graph-theoretical methods for analysing and optimising the behaviour of sets of synchronising parallel processes. We focus on processes that occur in Cyber-Physical Systems (CPSs), and are derived from a formal specification of such a CPS at the design level. These processes have strong requirements with respect to their timely execution and memory occupancy. We focus on optimising the execution time of the processes, taking into account that the memory requirements have to be met.

For a feasible implementation and resource-aware execution, it is advantageous and often necessary to combine sets of parallel processes that synchronise on certain actions. The graph-theoretical approach we have been developing in this thesis, clarifies and captures what we mean by combining sets of synchronising processes, and demonstrates how such combinations can be analysed and utilised in a systematic way.

1.1 Context

The creation of control software for CPSs is challenging, because the physical part of the CPS has great influence on the cyber part of the CPS, i.e. the interaction of the hardware and the software processes, and thereby possibly compromising the timely execution of these processes. Because we consider *software* processes only, in the sequel we mean by a *process* in the context of a CPS always a *software process*. The physical part of a CPS enforces restrictions on the tardiness of the control software (i.e. the cyber part of a CPS), which leads to far-reaching consequences. The deadlines of the processes have to be met because missing a series of deadlines in an arguably short period of time destabilises the CPS and leads inevitably to a catastrophe. Therefore, the processes are *not* allowed to be tardy.

The processes we consider are periodic. They are executed within every period, where the periods are repeating, equidistant time intervals of, for example, a 1 ms duration.

Each process has:

- a release time: this is the first time the process starts executing,
- a period: this is the time frame available for the process to execute,
- a relative deadline: this is the point in time with respect to the beginning of the current period, before which the process must have finished execution,
- a worst-case execution time: this is the maximum length of time the process may need to execute its task during a period.

A CPS comprising this kind of processes is a Periodic Hard Real-Time Control System (PHRCS).

We further restrict the CPSs to Embedded Control Systems (ECSs), like in robotic applications. To be able to design and maintain ECSs, we have to be able to reason about the behaviour of ECSs. Therefore the verification and validation[1] of ECSs are essential.

With respect to the behaviour of the CPS, in particular, for safety-critical systems, two issues are important: the safety property and the liveness property[2]. Whenever the safety property is not met, this leads to the situation where something bad, not envisioned by the designer, will happen. As an example, when in a computer-controlled, surgical robot a series of deadlines is missed by the actuators positioning the surgical knife, serious wounds can be inflicted onto the patient, which is obviously something bad. Important aspects of the liveness property are freedom from deadlock and freedom from starvation[3], where freedom from starvation implies freedom from deadlock. Especially deadlock avoidance is of interest because whenever a series of processes is deadlocked, they all miss their deadline with a catastrophe as a result. This real-time property, not being allowed to miss a deadline, enforces directly that the PHRCS must fulfil both the safety property as well as the liveness property with respect to timeliness. To be able to reason about timeliness, we need some kind of model representing the behaviour of the ECS in time that makes this reasoning possible. When there exists a model checker for such a model these real-time properties can be checked.

There are several ways to model the control part of the CPS, e.g. using formal specification languages like Communicating Sequential Processes (CSP) (Hoare, 1978), Finite State Processes (FSP) (Magee and Kramer, 1999), Temporal Logic of Actions (TLA+) (Lamport, 2002), Language Of Temporal Ordering Specification

[1] "Verification. The process of determining whether or not the products of a given phase of the software development cycle fulfil the requirements established during the previous phase. Validation. The process of evaluating software at the end of the software development process to ensure compliance with software requirements." (Boehm, 1984, page 75)

[2] "Safety properties are assertions of the kind 'nothing bad ever happens'" and "liveness properties are assertions of the kind 'something good eventually happens'".(Klapuri et al., 1999, page 70)

[3] "A situation ... in which all the programs continue to run indefinitely but fail to make any progress is called starvation." (Tanenbaum and Woodhull, 2005, page 89) and "A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause." (Tanenbaum and Woodhull, 2005, page 239)

(LOTOS) (ISO, 1987), or using object oriented methodologies like Real-Time Unified Modelling Language (RT-UML) (Gomaa, 2000; Douglass, 2014; Object Management Group (OMG), 2015), MARTE (Bran and Gérard, 2014; Object Management Group (OMG), 2015).

Among others, formal specification languages like CSP and TLA+ have model-checking support. A CSP model can be checked by a model checker like FDR (FDR, 2016). TLA+ is a mathematical approach using the temporal logic of actions (Lamport, 2002) and contains the TLC model checker. Techniques like RT-UML, do not have such model checkers. Schäfer et al. (2001) describe for the Unified Modelling Language (UML) a prototype tool for automatically verifying "whether the interactions expressed by a collaboration can be realized by a set of state machines." The tool does not check the safety property and liveness property. For these reasons the choice of a formal specification language is obvious.

Another reason to choose a formal specification language for our research, in particular, a process algebra, is that at the Robotics and Mechatronics group of the University of Twente a line of research for software for robotic applications is based on process algebra. This line of research has led to the software tool-chain Twente Embedded Real-time Robotic Application (TERRA) (Bezemer et al., 2012) and LUNA Universal Networking Architecture (LUNA) (Bezemer et al., 2011).

Designing ECSs using process algebras has two issues with respect to the real-time specifications of ECSs, which we are going to explain in more detail in the sequel:

- The usage of process algebras leads to fine-grained[4] concurrency of the constituent processes of the ECS. Therefore, ECSs comprise many short processes, where the process context switches may consume a considerable amount of the available processing power.
- Processes may have to wait for devices that produce information on which the control software has to act. As this is based on the rendezvous principle, it often happens that such a process is delayed, while it could perform other actions.

An approach to solve these two issues related to timeliness of ECSs is:

- Combining processes, thereby reducing the number of context switches, which decreases the execution time and thereby increases the performance of the ECS.
- Introducing a new modelling feature, which disconnects the processes involved in a rendezvous. By disconnecting the processes involved in the rendezvous, the end-to-end processing time of a set of processes can be reduced.

Both issues are dealt with during the design phase of the control software.

Our target systems are ECSs, like robotic applications, coming from the area

[4] A monolith of, for instance, one process is hard to design due to the complexity of such a process; the process has to meet all the cyber-part requirements of the CPS. A *divide and conquer* strategy leads to a division into many processes, where each process has to meet a subset of the requirements, and therefore to fine-grained concurrency.

of CPSs, where "a Cyber-Physical System (CPS) is a system of collaborating computational elements controlling physical entities" (Bagnato et al., 2014). Within these ECSs, our emphasis lies on the timeliness of the system. Because of the deadlines that have to be met, time-critical issues caused by a delay like latency and jitter have to be dealt with. But these kinds of delays are not always surmountable.

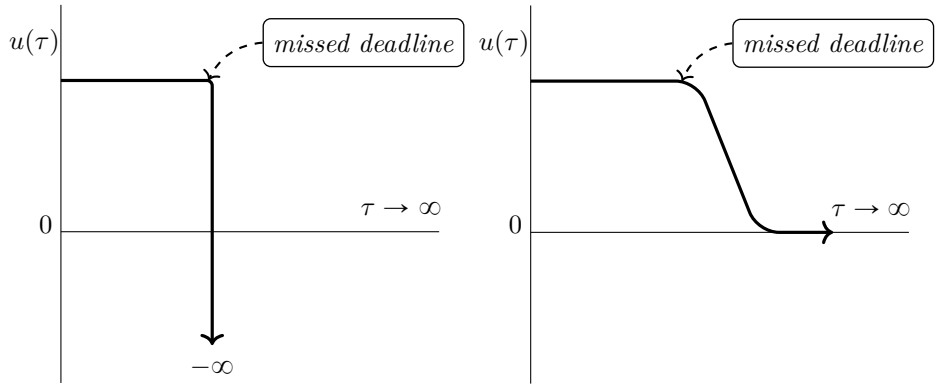
For our systems, where the processes often consist of reading a sensor value y, calculating a steering value resulting from a control law $f(y)$ and writing this value to an actuator, the deadline of a process is essential. When this series of actions is not performed in the available time-frame, the application will expose behaviour that violates the requirements. As an example, Surface-Mount Technology (SMT) Component-Placement Systems are used to place Surface-Mounted Devices (SMDs) onto a Printed-Circuit Board (PCB). Such precision-positioning equipment executes its task at high speed, with high precision. The lack of precision due to a deadline miss could misplace the component and thereby the PCB could be useless or arguably worse, could lead to a PCB that is used in some system, where it on an irregular basis produces catastrophic errors. As observed in Heemels and Muller (2006): "From a control point of view, time delay, consisting of the combination of both the latency and jitter, which includes computation times, communication delays and probably a reaction time of the sensors or actuators, is an undesired phenomenon that should be kept as small as possible. In control engineering, it is well known (Franklin et al., 2001) that these time-delays can degrade the performance of the controlled system and can even cause instability of this system."

We distinguish two kinds of timeliness for systems: hard real time and soft real time (Kopetz, 1997). Whenever a deadline is missed in a hard real-time system the consequences are catastrophic. For a soft real-time system after a deadline miss, the system may still execute, probably with less functionality, and may recover to normal operation.

The usability of a system after a deadline miss can be expressed in a utility function (Buttazzo, 2004) with respect to timeliness. For hard real-time systems, the utility function $u(\tau)$ drops to $-\infty$ instantaneous at a deadline miss (Figure 1.1a). For soft real-time systems, the utility function is a continuous function that, after a missed deadline, decreases to zero as time passes by (Figure 1.1b).

Among others, we have on the one hand that the real-time system might be fully event driven, where for each event a deadline is specified. On the other hand, the real-time system might be periodic, where for the tasks that execute within a certain period, deadlines are specified. We are describing systems that contain periodic processes but where processes can be event driven.

In fact, one may argue that our systems lie in between hard real-time systems and soft real-time systems. According to the observation in Heemels and Muller (2006), there can be a series of deadline misses before the system becomes unstable. Assuming that an unstable system is a catastrophe, the utility function drops to $-\infty$ after a series of deadline misses. The behaviour where a single deadline miss



(a) hard real-time systems **(b)** soft real-time systems

Figure 1.1: Utility function $u(\tau)$, adapted version of Buttazzo (2004), page 231.

is just a small decrease in utilisation, but a series of deadlines is catastrophic, is shown in Figure 1.2.

Buttazzo (2004) defines firm real-time as "executing a task after its deadline does not cause catastrophic consequences, but there is no benefit for the system, thus the utility function is zero after the deadline." In line with Buttazzo (2004), we define for periodic real-time systems, firm real-time as "infrequent deadline misses, less than k deadline misses in a given time frame of t s, will not be catastrophic for the system. The utility function $u(\tau)$ degrades to zero for one period after each deadline miss. But when at least k deadline misses occur within a given time frame of t s, this will lead instantaneously to a catastrophe and the utility function $u(\tau)$ degrades to $-\infty$ at the k^{th} deadline miss." Obviously, t and k are a consequence of the requirements of the application. The timing requirements of firm real-time systems are not as strict as the requirements of hard real-time systems. When we design a system that does not violate the hard real-time requirements, it will also not violate the firm real-time requirements. Therefore we will consider our systems as if they are hard real-time systems.

In CPSs we have a set of computational elements that control physical devices. The computational elements are collaborating to achieve some task, in our case a robotic application. From a software point of view, this is a PHRCS comprising computational elements represented by Periodic Hard Real-Time Control Processes (PHRCPs), controlling machines, i.e. external devices through sensors and actuators.

The behaviour of PHRCSs can be modelled using process-algebraic specifications (Schneider, 1999). We use process-algebraic specifications because they are formal manners to describe concurrent systems. A process-algebraic specification (for example, given in the form of a Finite State Process (FSP) (Magee and Kramer,

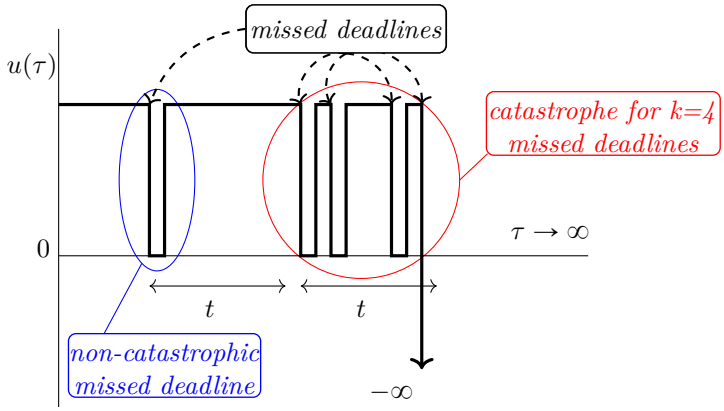


Figure 1.2: Utility function $u(\tau)$ for firm real-time systems.

1999)) can be implemented using Finite State Machines (FSMs). In essence, an FSM is a labelled and directed graph.

The processes described in an algebraic specification contain synchronous and asynchronous actions. Synchronous actions in processes lead to an overhead that is a result of context switches[5] in the system which executes the software representing the processes. Furthermore, the end-to-end processing time of a set of processes has to be reduced if the end-to-end processing time jeopardises the real-time requirements. As an example, this happens when one or more processes are waiting because they are involved in a rendezvous, no other process, apart from the processes involved in the rendezvous, is ready to execute, and one of the processes involved in the rendezvous is waiting, e.g. for some hardware action to finish.

Whenever a series of deadlines in an arguably short period of time of one or more processes is not met, this will lead to a catastrophe in the PHRCS and therefore a reduction of the overhead is essential.

Obviously, a designer can model the system in such a manner that overhead and excessive end-to-end processing time of a set of processes are avoided or reduced by the model. If this is required from the designer, it puts a burden on the designer and may lead to a system that does not fulfil all requirements or is error-prone. Taking away the need for performance on a design level gives the

[5]In the literature, there are several interpretations of a context switch. For example, according to Li et al. (2007) a "context switch refers to the switching of the CPU from one process or thread to another", whereas the interpretation of Pinto et al. (2012) is given by "When the application makes a system call it issues a software interruption that causes a context switch transferring the control to the kernel code, which then executes operations on behalf of the calling process." We follow the interpretation given by Li et al. (2007).

designer freedom of choice within the design alternatives without violating the hard real-time requirements.

The aim of our research is to improve the performance of concurrent systems described in a formal, testable manner, such that the system will behave as envisioned by the designer. Therefore, the focus of this thesis lies on the improvement of the performance of PHRCSs during the development of these systems, resulting in systems that have a reduced overhead by executing fewer context switches and less end-to-end processing time of a set of processes.

1.2 Problem description

Although software development has matured in the last decades, software is still designed by humans. The support of tooling during the design cycle has improved the quality of the software. Out of the range of open problems on design level (for example, inconsistencies in software models (Spanoudakis and Zisman, 2001)) we consider only the performance of the system with respect to the effort the designer has to put into the design. As soon as performance is an issue the designer has to address this performance issue and that may affect the design with, for example, less functionality as a result. A tool can address the performance issue and release the designer of taking performance into account during the design cycle.

Veldhuijzen (2009) measured that in control systems designed with the process algebra CSP context switches can lead to a considerable overhead of 20%.

Two reasons for the issue of superfluous context switches can be given for the overhead of such systems:

- the overhead due to the synchronisation of actions of processes, because of which the processes will execute in the proper order. This leads to two[6] extra context switches for every process except the first, that is participating in the synchronisation, and
- the overhead due to the passing of a series of variable values from one process to the other. This leads to two[6] extra context switches for each passing of a value of one process to the other.

Although both reasons can be dealt with on design level, the second reason is more of an implementation nature. As our focus lies on the design part of the software engineering process, the second reason is not taken into account.

Currently, we have no measurements showing deadline misses due to excessive end-to-end processing time provoked by the rendezvous of two or more processes. Still, we see this as a problem that could easily be solved with the introduction of a modelling feature disconnecting the processes involved in a rendezvous. This

[6]There are two extra context switches because, for every action of a process, Synchronisation Software (SyncSw) has to be informed of the action that a process wants to execute. Therefore there is one context switch performed by the Real-Time Operating System (RTOS) from a process to the Synchronisation Process and there is one context switch performed by the RTOS from the Synchronisation Process to the process.

feature can be used during the design of the control software. Therefore we have two issues to solve:

1. reducing the number of context switches during one period of execution of the PHRCS,
2. reducing the end-to-end processing time of a set of processes during one period of execution of the PHRCS.

1.3 Research questions

For the first issue of Section 1.2, we can reduce the number of context-switches by means of a transformation of a set of parallel processes specified in some process-algebraic formalism into graphs, after which we can use tools (graph products) that multiply these graphs. When transforming the results of the graph products back to processes, this will lead to a new set of possibly parallel processes for which the threads that are an implementation of these processes have fewer context switches.

As we are dealing with hard real-time systems, this improvement must occur for those situations where the involved processes are behaving in such a manner that they fully consume their worst-case execution time. Of course, we cannot neglect issues like power consumption. However, an improvement of the performance may give the processor the possibility to go into a power down mode and thereby saving energy. Still, we consider this a side effect, which should not obscure our goal, timeliness in PHRCSs.

For the second issue of Section 1.2, a solution would be the introduction of a new parallel operator together with new writing actions and reading actions which disconnect the processes performing these actions. Obviously, this will only lead to a performance improvement, if by using such an operator the timeliness is guaranteed, whereas the timeliness would be violated without the usage of this operator.

The two issues lead to three research questions:

1. How can the *number of context switches* be reduced for periodic hard real-time systems, which are developed using process algebras, by means of a graph-theoretical approach in such a manner that the *performance* of the system is improved?
2. What are the *algebraic and graph-theoretical properties of the graph-theoretical approach*, which are developed using process algebras, that reduce the number of context-switches for periodic hard real-time systems in such a manner that the performance of the system is improved?
3. How can the reduction of the *end-to-end processing time* of a set of processes during every period of any periodic hard real-time system, which is developed using the *process algebra CSP*, be achieved by means of an *extension* of the graph-theoretical approach of research question one?

1.4 Approach

From Section 1.2 it follows that the aim of our research is to improve the performance of PHRCSs by the reduction of the overhead due to the number of context switches and by reduction of the end-to-end processing time of a set of processes by disconnecting processes involved in a rendezvous.

Firstly, for synchronising actions we have to investigate whether such an improvement can be obtained by graph multiplication, of which we expect they will lead to fewer context switches.

Secondly, we are going to elaborate the graph-theoretical characteristics of the graph multiplication. This includes a study on the decomposition of graphs under the graph multiplication.

Finally, we extend input/output related actions as defined in the CSP process algebra for which we expect to achieve a further improvement with respect to reducing the end-to-end processing time of a set of processes. This implies that we have to extend the CSP process algebra with new operators that introduce asynchronous readers and writers. We study the impact on the graph multiplication, which will lead to an adapted version of this graph multiplication.

Because we expect that the changes for the extended input/output related actions are minor with respect to the graph-theoretical characteristics of the graph multiplication, the graph-theoretical characteristics of the extended graph multiplication are outside the scope of this research.

1.5 Outline

In Chapter 2, we describe the background for which we introduce a new graph product that we call the Vertex-Removing Synchronised Product (VRSP). We give the system architecture for which we created the VRSP and describe the relationship between processes and graphs. We introduce the optimisation of graphs by VRSP leading to fewer context-switches for the threads that are an implementation of the processes that represent these graphs. We elaborate the characteristics of process algebra as far as they are of importance for the VRSP. We introduce a new type of communication, half-synchronisation, which enriches process algebra.

In Chapter 3, based on our publication in the 35th International Conference on Communicating Process Architecture (Boode et al., 2013), we prove for finite, deterministic, labelled, acyclic, directed multigraphs, which contain synchronising arcs that the VRSP gives a performance gain. For these proofs we use several stages of the VRSP, each of which handles a different aspect of the graph product; the Cartesian Product, the Weak Synchronised Product, the Reduced Weak Synchronised Product and the final resulting product: the VRSP. The reason for using these intermediate products is that it eases the argumentation and proof.

In Chapter 4, based on our publication in the 36th International Conference on Communicating Process Architecture (Boode and Broenink, 2014), we present

a case study showing the advantage of the VRSP. We give the overall system architecture for which our optimisation is meant and present and compare the heuristics that will achieve this optimisation, based on our case study. We introduce a lattice for which each vertex represents a different outcome of the VRSPs of the graphs (a possible solution) representing the set of processes specified by the designer of the PHRCPs.

In Chapter 5, we give the combinatorial aspects of the VRSP. When the VRSP is associative the number of possible outcomes is given by the Bell number (B_n) series and when the VRSP is not associative this number is given by the Bessel number ($\tilde{B}_n$) series (Comtet, 1974).

In Chapter 6, we elaborate the graph-theoretical properties of the VRSP. These properties are based on the definitions of the binary operation on a set and the associativity and commutativity of this binary operation. We define the notion of consistency of pairs of graphs, based on the contraction of graphs with respect to a set of arcs.

In Chapter 7, we introduce two graph decomposition theorems which divide a graph G representing a process P into two graphs G_1, G_2 representing the processes P_1, P_2 in such a manner that the behaviour of the processes P_1 and P_2 during the parallel execution of P_1 and P_2 is identical to the behaviour of the process P. The two graph decomposition theorems are based on the contraction of sets of vertices.

In Chapter 8, based on our publication in the 38th International Conference on Communicating Process Architecture (Boode and Broenink, 2016) and our publication in the 39th International Conference on Communicating Process Architecture (Boode and Broenink, 2017), we describe a special case of input and output in CSP. We introduce the half-synchronous parallel alphabetised operator, which gives an ordering to actions with respect to different processes. To support the claim that the PHRCS has an improved performance, we give an example, where a performance gain is achieved for a PHRCS comprising a processor, an FPGA and two controllers.

We extend the half-synchronous parallel alphabetised operator by indexing the reading actions and allowing multiple asynchronous writing actions to the same channel. We elaborate this extension in a case study, the Controlled Emergemcy Stop (CES), showing a performance gain for the end-to-end processing time of a set of PHRCPs.

In Chapter 9, we summarise the results of our research and finish with the recommendations for further research.

2

About Process Algebra, Graph Theory, and Periodic Hard Real-time Control Systems

Our line of research lies in the intersection of Process Algebra, Graph Theory and Cyber-Physical Systems (CPSs). For this reason, we introduce in this chapter the process-algebraic topics and the graph-theoretical topics with respect to CPSs (in particular Periodic Hard Real-Time Control Systems (PHRCSs)) that are relevant for our research.

We use graph theory to solve a performance problem with respect to context switches in PHRCSs. Furthermore, we extend the process algebra CSP to reduce the end-to-end processing time of a set of processes engaged in a rendezvous. We incorporate this extension of CSP in our graph-theoretical solution of the context-switch problem in such a manner that the end-to-end processing time is indeed reduced. We study the effects that our solutions have on PHRCSs.

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

The processes are implemented as threads on the target system, where there is a one-to-one relationship between the set of processes and the set of threads (see Figure 2.1). On the target system, the result of the transformation of processes to graphs will be stored in a FSM like data structure in the threads. Whenever a process performs an action, the related thread will execute a state transition in its FSM. Processes can perform an action only if all processes that have that action in their alphabet are in a state which allows that action and all these processes will perform this action at the same time, atomically (atomicity is defined in Definition 2.2.4 on page 17). To support this synchronisation of actions, we need synchronisation software (the Synchronisation Software Server in Figure 2.1) that controls the transitions representing these actions. Due to the synchronisation requirements, the state transitions will lead to context switches. These context switches are the first one of the two causes of the performance problem we want to address.

Graph theory is used to combine the FSMs (leading to fewer processes and therefore to fewer threads) by which we expect to achieve fewer context switches. For example, if two threads want to perform a state transition based on an action a, for both threads two[1] context switches have to be performed, which adds up to four context switches. If the two FSMs of the threads are combined, only one thread containing the combined FSM has to perform a state transition based on action a, leading to only two[1] context switches.

The processes are designed using software tooling running on a general purpose computer (workstation). The threads run on a target system, the PHRCS. Therefore we start in Section 2.1 with a description of the relation between the design of the PHRCS using a general purpose computer and the execution of the PHRCS on a target system, i.e. the design level with no real-time requirements on the design process versus the execution level with hard real-time requirements on the PHRCPs. We explain in which manner the graph-theoretical approach is used on the general purpose computer and what the impact is of this approach on the target system.

In Section 2.2 we describe the relation between processes and graphs. The motivation for the new graph product that we are going to introduce is that it may lead to fewer context switches and, when the requirements on deadline or memory occupancy are violated, the process of graph multiplication can decide whether a combination of graphs exists that fulfil the requirements of the application with respect to deadline and memory occupancy. Furthermore, we describe the importance of a weak-bisimulation of processes with respect to the notion of a contraction of graphs. We show the equivalence of a weak-bisimulation on design level and consistency of graphs on execution level.

As we are going to manipulate the processes by means of transforming the related graphs, we address the algebraic characteristics of the graph multiplication operator, the VRSP, and the + operator in Section 2.3.

We finish in Section 2.4 with an introduction of the process-algebraic operators we use for the VRSP and the new process-algebraic operators we introduce for asynchronous writers and asynchronous readers and their impact on the VRSP. Using these new process-algebraic operators we reduce the end-to-end processing time of a set of threads, which is the second one of the two causes of the performance problem we want to address.

2.1 System Architecture

In Figure 2.1 the relation is given between the general purpose computer, on which the design and implementation takes place, and the target system, which executes the implementation.

[1] A context switch from the thread to the SyncSw and a context switch from the SyncSw to the thread.

During the design phase, the requirements of the application lead to a series of processes, represented by the FSP_1 to FSP_n in Figure 2.1.

In the implementation phase, each FSP_i is transformed into a thread that will be executed on the target system, $Thread_1$ through $Thread_n$ in Figure 2.1.

The threads form the logic of the application, which may change depending on the requirements of the application, whereas the Services, the Real-Time Operating System (RTOS) and the Device Drivers (DDs) are fixed, designed only once, for a specific hardware platform.

The threads are FSM-driven (represented by the graphs G_1 through G_n in Figure 2.1) and have to communicate with the Synchronisation Software (SyncSw). The SyncSw is responsible for the synchronisation of the actions that the threads want to execute.

The Hardware-Dependent Software (HDS) and the Algorithmic Software (AlgSw) are FSM-driven as well. In general, they will have all hardware or algorithmic actions as labels in their alphabet. For example, if a thread can execute an action *motor.X.go*.10, motor X will make 10 revolutions per second, the HDS FSM will have a *motor.X.go*.10-labelled transition. Because both the thread and the HDS are able to execute *motor.X.go*.10, the SyncSw will notify both of this action. The HDS will execute the software implemented for *motor.X.go*.10 and will send the appropriate commands to the DD controlling *motor.X*. The HDS and the thread will make a state transition based on the *motor.X.go*.10-action. In the

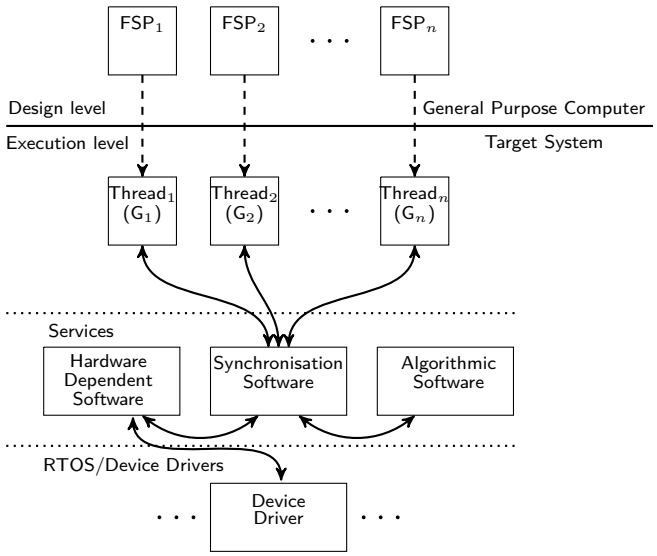


Figure 2.1: System Architecture.

architecture, shown in Figure 2.1, every action of a process on design level leads to a series of context switches for the threads; the threads, sending their information on the action to the SyncSw, the SyncSw deciding whether any message has to be send to these threads and the threads receiving the acknowledgement so they can execute the action (Boode and Broenink, 2014).

2.2 Processes and Graphs

Processes can be represented by directed graphs. By multiplication of these graphs according to the VRSP we are going to introduce, we obtain a graph which represents a process that may have fewer context switches than the original set of processes. For this purpose we have developed a VRSP, a binary relation with symbol $\boxslash$, (Boode et al., 2013; Boode and Broenink, 2014; Boode et al., 2015). In Figure 2.2 we give the expected behaviour of the processes with respect to execution time and memory occupancy when we optimise by multiplication of the graphs representing these processes by the VRSP. The set of parallel processes $P_1||\cdots||P_n$ is represented by the set of graphs $\sum_{i=1}^{n} G_i$. The process P, which is strongly bisimilar (Definition 2.2.1 on page 15) to $P_1||\cdots||P_n$, is represented by $\bigboxslash_{i=1}^{n} G_i$. Because the VRSP operates on two graphs, every VRSP of two graphs will reduce the number of graphs representing the process-algebraic specification by one and therefore one process less on the *Number of Graphs* abscissa.

Graph multiplication according to the VRSP is (worst-case) exponential with respect to memory occupancy if the graphs do not synchronise. This happens when the alphabets of the processes represented by these graphs do not share actions. By selecting processes that synchronise heavily, the memory occupancy of the multiplication of these processes can decrease. This happens (best case) when the alphabets of the processes represented by these graphs are identical. How the multiplications of a set of graphs will grow with respect to memory occupancy depends on the degree of synchronisation of the chosen graphs.

In Figure 2.2 the blue line shows a set of graphs that synchronise heavily. At the right side of the blue line, this is shown by a decreasing memory occupancy. The middle of the blue line shows that the multiplied graphs do not synchronise heavily any-more and the left side of the blue line shows that the last multiplications have little synchronisation.

As a contrast, the red line shows a set of graphs that do not synchronise heavily from the beginning (the right side) of the multiplications. While multiplying, the graphs synchronise continuously less. Note the logarithmic scale for the memory occupancy.

The graph multiplication is necessary when the threads cannot meet the hard real-time requirements of the application; a series of deadline misses in an arguably short period of time is a catastrophe. When multiplying the graphs, the memory occupancy may grow, even exponentially. Therefore it is possible that *no* set of

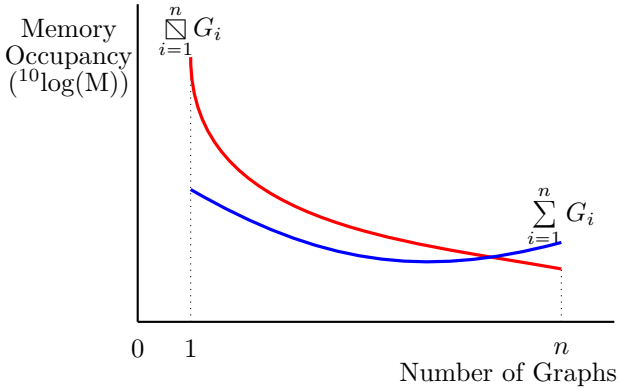


Figure 2.2: Maximal synchronisation (blue line) versus minimal synchronisation (red line).

multiplied graphs exists that fulfil the requirements of the application with respect to the deadline and memory occupancy. In this case, either the system has to be redesigned or more powerful hardware has to be chosen.

The relation between concurrent processes and graphs in relation to bisimulation (Definition 2.2.1 and Definition 2.2.2) and graph multiplication is shown in Figure 2.3 on page 17. Our definitions of a strong bisimulation and weak bisimulation are based on Milner (1989).

Definition 2.2.1.
Let $\mathcal{P}$ be a set of states and let Act be a set of actions. Then a binary relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is a strong bisimulation if $(P, Q) \in S$ implies, for all $\alpha \in Act$,

(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some $Q', Q \xrightarrow{\alpha} Q'$ and $(P', Q') \in S$

(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some $P', P \xrightarrow{\alpha} P'$ and $(P', Q') \in S$

Milners definition implies that for two processes that are strongly bisimilar, the set of traces of one process is identical to the set of traces of the other process, but not vice versa. We denote two strong-bisimilar processes P_1, P_2 as $P_1 \sim P_2$.

The kind of bisimilarity defined in Definition 2.2.1 is too strong when we want to describe the behaviour of parallel processes, because processes that are strongly bisimilar are in fact indistinguishable. We want the processes to have on the one hand displaying unique, therefore asynchronous, behaviour and on the other hand displaying behaviour synchronously with some of the other processes. If we consider the asynchronous behaviour of a process as silent actions τ for the other processes, we can use the definition of a weak bisimulation, given in Definition 2.2.2.

For a set of states $\mathcal{P}$ with $\alpha \in Act$, $X, X', Y, Y' \in \mathcal{P}$, we write $X \stackrel{\alpha}{\Rightarrow} Y$ if and only if

- if $\alpha \neq \tau$, we have $X \stackrel{\tau^*}{\rightarrow} X' \stackrel{\alpha}{\rightarrow} Y' \stackrel{\tau^*}{\rightarrow} Y$
- if $\alpha = \tau$. we have $X \stackrel{\tau^*}{\rightarrow} Y$, where τ^* stands for a (possibly empty) sequence of τ-labelled transitions.

Then a weak bisimulation is defined as follows:

Definition 2.2.2.
Let $\mathcal{P}$ be a set of states and let Act be a set of actions. A binary relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is a weak bisimulation if $(P, Q) \in S$ implies, for all $\alpha \in Act$,

(i) Whenever $P \stackrel{\alpha}{\rightarrow} P'$ then, for some $Q', Q \stackrel{\alpha}{\Rightarrow} Q'$ and $(P', Q') \in S$

(ii) Whenever $Q \stackrel{\alpha}{\rightarrow} Q'$ then, for some $P', P \stackrel{\alpha}{\Rightarrow} P'$ and $(P', Q') \in S$

Using the weak bisimulation we can define consistency of processes. We want two processes to be consistent if each process on its own is able to exhibit the same behaviour as when the process is part of the two processes in parallel. In the sense of observable behaviour, this means that whenever a process is not engaging in an action of another process, it does not have that action in its alphabet; from a synchronisation point of view, that action is not observable by the process. Hence, we redefine the silent, not observable, action τ as an asynchronous action for any process P. Then consistency of processes is defined in Definition 2.2.3.

Definition 2.2.3.
Let τ represent any asynchronous action of either the process P or the process Q. Then the processes P and Q are consistent if and only if they are weakly bisimilar, denoted as $P \approx Q$.

For graphs, we have defined the notion of *consistency* of graphs, denoted as $\div$, in Chapter 5, on page 73, which is based on the contraction of graphs, in such a manner that our interpretation of a weak-bisimulation on design level is equivalent to *consistency* on execution level.

Next to the weak-bisimulation, we also need the strong-bisimulation because whenever a set of parallel processes P behaves in a strong-bisimular fashion to one process Q, P is consistent with Q. As an example, in Figure 2.3 we show the transformations, where G_1, G_2 are consistent graphs, $P_1||P_2$ and $P_{1,2}$ are strongly bisimilar and P_1 and P_2 are weakly bisimilar. Threads executed on our target system will provoke a context switch whenever an action is executable. In such a system synchronisation software must exist that is responsible for the synchronisation of threads. These threads can only execute a certain action if all threads in the system for which the related process has this action in its alphabet (visible to other processes) are able to execute the action.

On the level of design, processes synchronise over an action atomically. For this synchronisation, we use the definition of atomicity of Lomet (1976) (Definition 2.2.4).

$$\begin{array}{ccccc} P_1 \approx P_2 & \Leftrightarrow & (P_1 || P_2 & \sim & P_{1,2}) \\ \Updownarrow & & & & \Updownarrow \\ G_1 \div G_2 & \Leftrightarrow & (G_1 + G_2 & \div & G_1 \boxtimes G_2) \end{array}$$

Figure 2.3: Relation between weak bisimilarity and strong bisimilarity of processes, and consistency of graphs.

On the implementation level, this requires software that controls the threads for which the related process is synchronising over an action. This will not be atomic in the sense that the atomic action over which processes synchronise, cannot be executed by the threads at the same moment in time. The execution of the action by one thread will follow the execution of the action by the other thread. In the meantime, the threads can be interrupted by other threads (not involved in the synchronisation). The synchronisation software has to assert that the execution of the synchronised action by the involved threads is atomic as far as these threads are concerned.

Definition 2.2.4. *(Lomet, 1976) Actions are atomic if they can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.*

We use Lomets definition of atomicity from the perspective of a process-algebraic (is design) level.

The components under consideration are PHRCPs, with identical priorities, deadlines and release times. They differ only in their behaviour which may lead to different computation times.

2.3 Algebraic and Graph Theoretical Characteristics

The characteristics of addition and multiplication in group theory deal with, for a given operator and a given set, idempotency, distributivity, associativity, commutativity and invertibility. From the definition of the VRSP given in Section 6.1.2 it is obvious that the VRSP is idempotent, commutative and not distributive. We give conditions and prove in Section 6.4 for which the VRSP is associative. Because invertibility has no meaning in process algebra, we can disregard invertibility of graphs under the VRSP.

In process algebra, it is often claimed that the parallel operator is associative (Hoare, 1978; Magee and Kramer, 1999; Schneider, 1999). Hoare (1978) shows this by mentioning that for the choice operator "A process which first does x and then makes a choice is indistinguishable from one which first makes the choice and then does x" followed by the remark that for the same reasons the parallel operator is

associative. So in Hoare's view, the usage of an operator leads to distinguishing (or not) between processes, whereas in our view it should be isomorphic (or not).

In a way, Roscoe (2010) solves this by creating an alphabetised parallel operator ${}_x||_y$, which is associative if the alphabets of all the processes are taken into account and therefore the processes are allowed to change their behaviour depending on the context in which they operate, $((A_x||_y B)_{x\cup y}||_z C = A_x||_{y\cup z}(B_y||C_z))$. Furthermore, Roscoe remarks that $(P \underset{X}{||} Q) \underset{X}{||} R = P \underset{X}{||} (Q \underset{X}{||} R)$ is weak (in that both interfaces are the same) associative and that it is hard to construct a universally applicable associative law for $(P \underset{X}{||} Q) \underset{Y}{||} R = P \underset{X}{||} (Q \underset{Y}{||} R), X \neq Y$.

As another example, Magee and Kramer (1999) just mention that their parallel operator is associative.

In process algebra the behaviour of two parallel processes depends on the other processes in the parallel execution. In fact, one may argue that the parallel operator is an n-ary operator instead of a binary operator.

We have the classic view on associativity of a binary operator $\circ$ on a set S. First of all, S has to be closed under $\circ$. Secondly, $(a \circ b) \circ c = a \circ (b \circ c)$ for all $a, b, c \in S$. Therefore, for our graph product, each multiplication of two graphs must be unique and must not depend on other graphs in the multiplication. But this is as well not the case for the parallel operator in process algebra.

2.4 Operators in Process Algebra and Graphs

The transformation of process algebras into graphs is well known. As an example Vrancken (1997) shows transformations of processes to graphs for ACP$^\epsilon$ (Bergstra and Klop, 1989). For our goal, PHRCPs, we are only interested in a subset of the operators of a process algebra; the choice and parallel operator. We use FSP (Magee and Kramer, 1999) and CSP (Hoare, 1978) as the process algebras to formulate our examples and case studies.

The rendezvous protocol is implicit in process algebra, as two (or more) processes are only allowed to have a transition with respect to a certain label if all processes containing this label in their alphabet are in a state where they can perform this transition. These processes will perform this transition simultaneously. Such a transition is performed atomically. Therefore, the rendezvous protocol is synchronous.

We introduce half-synchronisation as a form of communication in between synchronisation and a-synchronisation.

3

Minimising the Length of a Graph

This chapter is based on our paper presented at the CPA 2013 conference (Boode et al., 2013).

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. (2007) showed that the average cost of a context switch varies from 3.8 μs [1] to over 1 ms [2]. Veldhuijzen (2009) showed that the cost of a context switch is on average 7.7 μs [3]. Clearly, these figures depend on the hardware and software being used. 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. We restrict these configurations to robotic applications. 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

[1] Measured on a 2.0 GHz Intel Pentium Xeon processor, running under the Linux 2.6.17 kernel with Redhat 9.

[2] For context switches, Li et al. (2007) distinguish 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 pipeline, but also kernel code that has to execute. Indirect costs include cache misses caused when there is a context switch to a process whose cache lines have been reused. Such costs may degrade performance in a significant way.

[3] Measured on a 560 MHz Intel Pentium IV processor, running under the QNX operating system.

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 et al. (2007), the 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 (Hoare, 1985)); on the other hand, there is the execution of processes representing the specification, on a computer system, running under an operating system.

At the 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.

At the 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 the 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 connected, finite, labelled, acyclic, directed multigraph[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 label, 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 that we are going to introduce in this chapter.

[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 point (used in intermediate stages of the graph transformations described later).

Via a design methodology, a process specification has to be transformed into a program. We insert into this transformation three steps, of which this thesis 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.

Before we specify the model that we use to analyse the performance of periodic real-time processes, we introduce the necessary graph-theoretical and process-algebraical terminology in Section 3.1. In Section 3.2, we introduce periodic real-time processes as finite, labelled, acyclic, directed multigraphs, and we present an overview of existing synchronised products. In Section 3.3, 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 3.4 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 3.5 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 3.6 the Vertex-Removing Synchronised Product (VRSP) is introduced. We prove that the longest path length of the VRSP of a set of graphs is at most the longest path length of the disjoint union of these graphs. We present sufficient conditions for which the application of the VRSP leads to a reduction of the longest path length. We finish with our conclusions in Section 3.7.

3.1 Terminology

We use Bondy and Murty (2008) and Schneider (1999) for terminology and notation on graphs and processes not defined here and consider finite, labelled, acyclic, directed multigraphs only.

So, if we use G to denote a graph, we will always mean a finite, labelled, acyclic, directed multigraph. Thus G 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 $l(a) = l(b)$,

[6]We shall also use the notation $V(G)$ and $A(G)$ to denote the vertices and arcs of a graph G.

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 G is a sequence of distinct vertices $v_1v_2 \ldots v_k$ of G such that $v_jv_{j+1} \in A$ for $j = 1, \ldots, k-1$. The length of a path $v_1v_2 \ldots v_m$ is defined as $\sum_{i=1}^{m-1} t(v_iv_{i+1})$. A directed path defines a *total ordering* on its arcs: $v_1v_2 < v_2v_3 < \ldots < v_{k-1}v_k$.

A directed cycle is a directed path $v_1v_2 \ldots v_k$ together with an additional arc v_kv_1, and is denoted by $v_1v_2 \ldots v_kv_1$. An acyclic graph does not contain any directed cycles.

In general, a finite, labelled, acyclic, directed multigraph G consists of several components, where each component, G_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, $\ell(G_i)$ is defined as the maximum length taken over all directed paths in G_i. For the whole graph, which corresponds to a parallel set of sequential processes that each must run to completion, the maximum path length, $\ell(G)$, is the sum of all the individual $\ell(G_i)$. 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 ordered in some directed path within the graph.

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

Time and processes are thoroughly described in CSP (for example, by Schneider (1999)). 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, the actions of the process constitute the process alphabet set A_P, which consists of labels. A label in a process is identical to a label in a graph: both are identical strings of characters with an identical associated value.

Whenever in a certain state s of a process P a choice exists out of n actions, where an action is labelled (l_i, t_i), into $m \leqslant n$ states $s_1, \ldots, s_m$, this is represented in the graph G by the vertex $u \in V(G)$ representing the state s, the vertices $v_1, \ldots, v_m \in V(G)$ representing the states $s_1, \ldots, s_m$, respectively, and n arcs $a_1, \ldots, a_n \in A(G)$, with tail u and head v_j, $j \in \{1, \ldots, m\}$ and with labels $\lambda(a_i) = (l_i, t_i), i = 1, \ldots, n$.

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 G_i whose label does not occur on an arc of another component G_j, the corresponding action is not blocked from execution.

One of the cases in which a graph G representing a set of parallel processes is said

to be ill-defined or inconsistent is when this set of processes contains a deadlock. In Definition 6.2.1 on page 73, we define consistency of graphs in a broader context.

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

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

The Cartesian product $G_1 \square G_2$ of G_1 and G_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 G_1 and G_2, respectively. For $k \geqslant 3$, the Cartesian product $G_1 \square G_2 \square \ldots \square G_k$ is defined recursively as $((G_1 \square G_2) \square \ldots) \square G_k$.

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

3.2 Periodic Real-time Processes as Labelled 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 PHRCP must not contain loops with an unbounded number of iterations; otherwise it may happen that a deadline is missed due to this unbounded number of iterations in such a loop consuming an unbounded amount of execution time. Therefore, a process must be acyclic and can be defined as a labelled, directed, acyclic graph. If the process specification contains cycles, these cycles must be bounded with a fixed upper limit known during the design phase of the PHRCS. In such a case the cycles can be unfolded leading to an acyclic directed graph.

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

Several products of graphs for combining a set of graphs into one graph have been defined in the literature, like strong products, synchronised products, etc. None

[7] There are no *divergences*[8] as the processes being combined are finite (repeated *periodically* by the real-time application).

[8] "A divergence is a finite trace during or after which the process can perform an infinite sequence of consecutive internal actions."(FDR, 2016)

of these products is sufficient for our purposes. The strong product as defined by Bondy and Murty (2008) is a 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. (2005), Caucal and Hassen (2008), Hammal (2007) and Wöhrle and Thomas (2004). The definition by Hammal (2007) 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 by Caucal and Hassen (2008) 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 product of graphs we have in mind 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. (2005) stems from Input Output Symbolic Transition Systems and turns out to be almost similar to our product, although the terminology is different. Even the definition by Wöhrle and Thomas (2004) does not fit our needs, although this product preserves the order as shown in Figure 3.1 (the dashed arc is the synchronising arc). In their approach it is possible for the synchronised product of two weakly connected graphs (shown on the left of Figure 3.1) to contain again two or more weakly connected graphs (shown on the right of Figure 3.1, where the diamond-shaped component and the isolated vertex represents states and transitions unreachable according to the synchronisation rules).

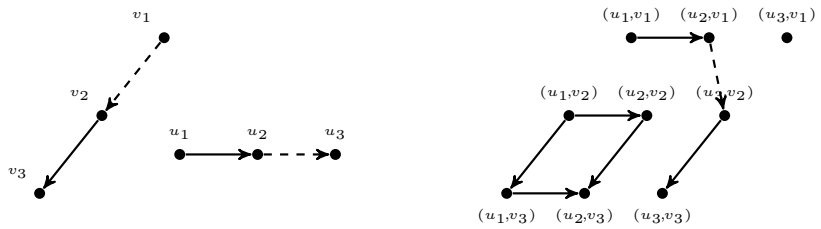


Figure 3.1: Synchronising product according to Wöhrle and Thomas.

At first sight, the Cartesian product of graphs seems to be a good way to express the combination of the corresponding parallel processes. However, this product represents the interleaved execution of the processes and does not take into account the synchronisations required by the processes which must execute synchronising actions atomically. Therefore, we propose a modification of the synchronised product by Wöhrle and Thomas (2004) that will be developed in a number of steps. Figure 3.2 shows an example of five graphs, where dashed arcs represent synchronising actions.

The example shows five steps, where the VRSP is built from a set of graphs.

These five steps are elaborated in Sections 3.3 through 3.6, where also the formal definitions of the corresponding products are given.

In the upper left of Figure 3.2, G_1 and G_2 represent two processes with one synchronising arc (the dashed arc in both graphs); $G_1 + G_2$ denotes the disjoint union of these two graphs.

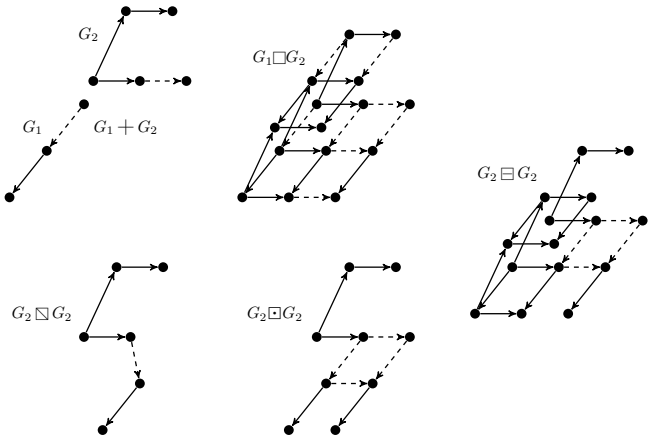


Figure 3.2: Transformations from parallel ($+$) via the Cartesian product ($\square$), the weak synchronised product ($\boxminus$), the reduced weak synchronised product ($\boxdot$) to the VRSP ($\boxtimes$).

In the upper middle of Figure 3.2, $G_1 \square G_2$ is the Cartesian product of the graphs G_1 and G_2. 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 will be explained in more detail in Section 3.3.

On the right of Figure 3.2, $G_1 \boxminus G_2$ is the weak synchronised product of the (original) graph $G_1 + G_2$. It is derived from $G_1 \square G_2$ 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 of two-dimensional parallelograms (one dimension for each original process engaging in the action). If there is a deadlock in the system, this will appear as vertices with no out-flowing arcs. This will be explained in more detail in Section 3.4.

In the lower middle of Figure 3.2 $G_1 \boxdot G_2$ is the reduced weak synchronised product of G_1 and G_2. It is derived from $G_1 \boxminus G_2$ 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 will be explained in more detail in Section 3.5.

Finally, in the lower left of Figure 3.2 $G_1 \boxtimes G_2$ is the VRSP of G_1 and G_2. This collapses the common action parallelograms into single action arcs across the diagonal, leaving the two isolated vertices. The same iterative process from the third step (for removing vertices with no in-flowing arcs and their out-flowing arcs) cleans up. This will be explained in more detail in Section 3.6.

3.3 The Cartesian Product of a Set of Parallel Processes

To visualise all our transformations we use a simplified version of an untimed example by Oguz et al. (2012) given in Listing 3.1, shown in Figure 3.3. The

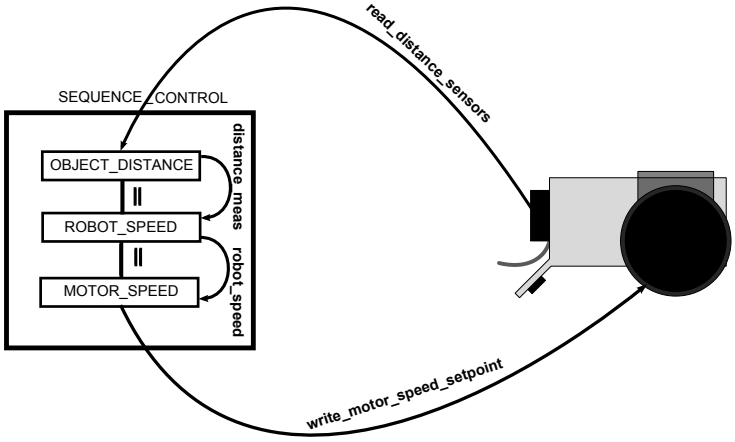


Figure 3.3: Untimed sequence control processes of a mobile robot

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 `Skip`s 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 Appendix I, Listing 1 and the related graph transformation in Figure 1 give a (slightly) more complex, and interesting example.

The process `SEQUENCE_CONTROL` in Listing 3.1 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 3.4). 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, e.g. (`read_distance_sensors`, 120 μs) will become *rds*. As before, a dashed or dotted arc represents a synchronising arc. 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 in Listing 3.2.

$$
\begin{aligned}
OBJECT\_DISTANCE &= read\_distance\_sensors \rightarrow compute\_object\_distance \rightarrow \\
&\quad distance\_meas \rightarrow Skip;\\
ROBOT\_SPEED &= distance\_meas \rightarrow compute\_robot\_speed \rightarrow robot\_speed \rightarrow Skip;\\
MOTOR\_SPEED &= robot\_speed \rightarrow compute\_motor\_speed \rightarrow write\_motor\_speed\_setpoint \\
&\quad \rightarrow Skip\,;\\
SEQUENCE\_CONTROL &= (OBJECT\_DISTANCE \parallel ROBOT\_SPEED \parallel MOTOR\_ \\
&\quad SPEED); SEQUENCE\_CONTROL;
\end{aligned}
$$

Listing 3.1: Description of the *SEQUENCE_CONTROL* process.

$$
\begin{aligned}
MS = \ & \big\{(V(G_1), A(G_1), \{\lambda(a)|a \in A(G_1)\})\big\}\\
& \big\{\big(\{v_1, v_2, v_3, v_4\}, \{v_1v_2, v_2v_3, v_3v_4\}, \{(v_1v_2, rs), (v_2v_3, cms), (v_3v_4, wmss)\})\big\}\\
RS = \ & \big\{(V(G_2), A(G_2), \{\lambda(a)|a \in A(G_2)\})\big\}\\
& \big\{\big(\{v_5, v_6, v_7, v_8\}, \{v_5v_6, v_6v_7, v_7v_8\}, \{(v_5v_6, dm), (v_6v_7, crs), (v_7v_8, rs)\})\big\}\\
OD = \ & \big\{(V(G_3), A(G_3), \{\lambda(a)|a \in A(G_3)\})\big\}\\
& \big\{\big(\{v_9, v_{10}, v_{11}, v_{12}\}, \{v_9v_{10}, v_{10}v_{11}, v_{11}v_{12}, \}, \{(v_9v_{10}, rds), (v_{10}v_{11}, cod), (v_{11}v_{12}, dm)\})\big\}
\end{aligned}
$$

Listing 3.2: Definition of the graph representing the *SEQUENCE_CONTROL* process.

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 3.4, it can 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, acyclic, directed multigraphs.

In the Cartesian product $G_1 \square G_2$ of G_1 and G_2 we distinguish between two types of arcs. Arcs of type G_1 (type G_2) are between pairs $(v_1, w_1) \in V(G_1 \square G_2)$ and $(v_2, w_2) \in V(G_1 \square G_2)$ with $v_1v_2 \in A(G_1)$ and $w_1 = w_2$ (with $v_1 = v_2$ and $w_1w_2 \in A(G_2)$), so arcs of type G_1 and type G_2 correspond to (are in fact copies of) arcs of G_1 and G_2, respectively.

For $k \geqslant 3$, the Cartesian product $\square_{i=1}^{k} G_i = G_1 \square G_2 \square \ldots \square G_k$ is defined recursively as $((G_1 \square G_2) \square \ldots) \square G_k$.

If no ambiguity can arise, we write $\square G_i$ for $\square_{i=1}^{k} G_i$. In this product of k directed graphs, we distinguish between arcs of type G_i for $i = 1, \ldots, k$, analogously as for the case $k = 2$.

Note that in case the G_i are labelled, the labels of the arcs of type G_i in $\square G_i$

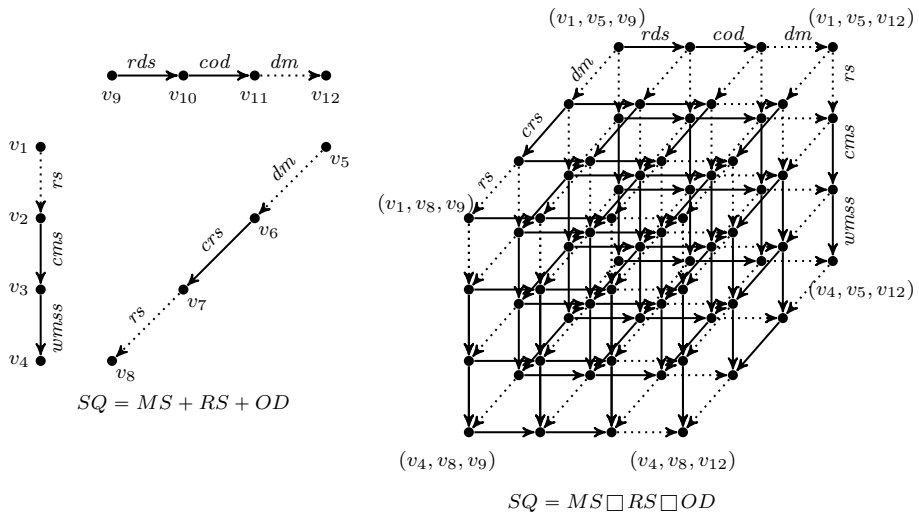


Figure 3.4: Sequence control processes of a mobile robot, from + to $\square$.

correspond to the labels of the arcs of G_i: each copy of an arc $a \in A(G_i)$ in $\square G_i$ has label $\lambda(a)$. If G_i is a multigraph an arc $a \in A(G_i)$ can appear more than once in G_i, but in that case, the copies of a in G_i have distinct labels, so each of the copies can be identified by its label. In $\square G_i$ similarly, we can distinguish the copies of a by their labels $(\lambda_1(a), \lambda_2(a), \ldots)$.

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 Bang-Jensen and Gutin (2008).

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 G_i are acyclic, directed multigraphs. For the moment, we disregard the labels, so in the following paragraphs the length of a directed path is just the number of arcs.

For each G_i we define S_0^i, also called the source of G_i, to denote the set of vertices with in-degree 0 in G_i, S_1^i the set of vertices with in-degree 0 in the graph obtained from G_i by deleting the vertices of S_0^i and all arcs with tails in S_0^i, 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. We also define the sink of G_i as the set of vertices with out-degree 0 in G_i. As in the acyclic ordering, this ordering implies that arcs of G_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 G_i, and that t_i is the length of a longest directed path in G_i, so $t_i = \ell(G_i)$. In fact, all longest directed paths of G_i have their starting vertex in S^i_0 and their terminating vertex in $S^i_{t_i}$. If a vertex $v \in V(G_i)$ is in the set S^i_j in the above ordering, we also say that v is at level j in G_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 $uv \in A(G_i)$. 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 $G_1, G_2, \ldots, G_k$ 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 G_i, $i = 1, 2, \ldots, k$. In fact, we prove the stronger statement that each longest path P in $\square G_i$ corresponds to longest paths in all G_i, in the sense that P contains exactly one copy of each of the arcs of a longest path Q_i in G_i, $i = 1, 2, \ldots, k$. We say that P is the interleaved concatenation of these Q_i.

Lemma 3.3.1. *Let G_i be an acyclic graph for $i = 1, 2, \ldots, k$. Then $\square G_i$ is acyclic and every longest path in $\square G_i$ is the interleaved concatenation of longest paths Q_i in G_i, $i = 1, 2, \ldots, k$. In particular, $\ell(\square G_i) = \ell(G_1) + \ell(G_2) + \ldots + \ell(G_k)$.*

Proof. First note that it suffices to prove the statements for $k = 2$, since for integers $k \geqslant 3$, $G_1 \square G_2 \square \ldots \square G_k$ is $((G_1 \square G_2) \square \ldots) \square G_k$, hence $G'_1 \square G'_2$, and the result follows by induction. So we want to prove that $G_1 \square G_2$ is acyclic and that every longest path in $G_1 \square G_2$ is the interleaved concatenation of longest paths Q_1 and Q_2 in G_1 and G_2, respectively.

It is easy to show that there exists a path in $G_1 \square G_2$ that is the interleaved concatenation of two longest paths Q_1 and Q_2 in G_1 and G_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 key to the remaining part of our proof are the following observations on paths in $G_1 \square G_2$. Consider a (longest) path P in $G_1 \square G_2$ that starts with a subpath Q_1 of type G_1 arcs only, followed by a subpath R_1 with a first arc of type G_2 (and with R_1 possibly containing arcs of type G_1 as well). Then Q_1 corresponds directly to a path Q'_1 in G_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 $V(G_2)$), 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 $x \in V(G_1)$. The vertex pairs corresponding to the vertices of R_1 keep this first coordinate x as long as the arcs are of type G_2. In case these arcs are followed by an arc of type G_1, this arc of type G_1 corresponds to an arc in G_1 starting from x. So, all the subsequent subpaths of P with only arcs of type G_1 correspond directly to paths $Q'_1, Q'_2, \ldots$ in G_1, and similarly all the subsequent subpaths of P with only arcs of type G_2 correspond directly to paths $R'_1, R'_2, \ldots$ in G_2. Moreover, there is an arc in G_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 G_2. By symmetry, the same observations can be made if the path P starts with an arc of type G_2, and contains arcs of both types.

To prove that $G_1 \square G_2$ 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 G_1 or G_2, contradicting our assumption that G_1 and G_2 are both acyclic. Assuming, without loss of generality, that the first $k \geqslant 1$ arcs of C are of type G_1, p_1 is the first vertex of a path $Q'_1 = p_1 p_2 \ldots p_{k+1}$ in G_1, with the corresponding subpath of C in $G_1 \square G_2$ consisting of vertex pairs (p_i, q_1), $i = 1, \ldots, k+1$. Then the first arc of type G_2 in C we encounter is from (p_{k+1}, q_1) to (p_{k+1}, q_2) for some $q_1 q_2 \in A(G_2)$, and so on, so q_1 is the first vertex of a path R'_1 in G_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 G_1 and G_2, respectively. Since by the above argumentation all the subpaths of type Q'_i are connected, this implies that G_1 contains a cycle, a contradiction. This proves that $G_1 \square G_2$ is acyclic.

Suppose now that P is a longest path in $G_1 \square G_2$. Assume that P has length $\ell(G_1 \square G_2) > \ell(G_1) + \ell(G_2)$. Using the above argumentation and the fact that $G_1 \square G_2$ is acyclic, the two paths Q and R formed by the Q'_i in G_1 and the R'_i in G_2, respectively, together have length $\ell(G_1 \square G_2) > \ell(G_1) + \ell(G_2)$, but this contradicts that the length of Q is at most $\ell(G_1)$ and the length of R is at most $\ell(G_2)$. Together with the above arguments, this shows that P has length exactly $\ell(G_1) + \ell(G_2)$ and that P is the interleaved concatenation of the two longest paths Q and R in G_1 and G_2, respectively. This completes the proof of Lemma 3.3.1. □

Remark 3.3.2. The expression in Lemma 3.3.1 on the length of longest paths in the Cartesian product is not valid if we drop the condition that each of the G_i is acyclic. It is easy to present counterexamples. For instance, consider $G_1 \square G_2$, where G_1 consists of two arcs connecting 3 unique vertices and G_2 has two arcs between two vertices, but in opposite directions (i.e. a cycle). As can be observed in Figure 3.5, the longest path lengths of G_1 and G_2 are 2 and 1, respectively[9], making their sum 3. However, $G_1 \square G_2$ contains a directed path of length 5.

[9]The vertices in a directed path must be distinct - see Section 3.1.

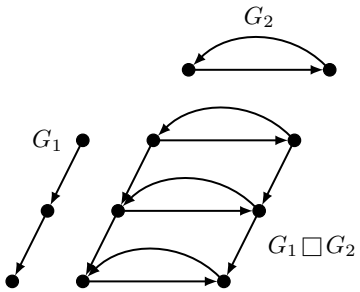


Figure 3.5: Cyclic graph counterexample.

Remark 3.3.3. 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 $(v_1, v_2, \ldots, v_k) \in V(\square G_i)$ we define the level vector $(f_1, f_2, \ldots, f_k)$, in which f_i denotes the level of vertex v_i in G_i. Then the vertices with in-degree 0 in $\square G_i$ are precisely all vertices with level vector $(0, 0, \ldots 0)$, whereas level vector $(t_1, t_2, \ldots, t_k)$ with $t_i = \ell(G_i)$ corresponds to all vertices that are terminals of some longest path in $\square G_i$. For each integer vector $(x_1, x_2, \ldots, x_k)$ with $0 \leqslant x_i \leqslant f_i$, there exists a vertex in $\square G_i$ with this level vector, and if $x_i < f_i$, there also exists an arc of type G_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 that there are several longest paths in $\square G_i$, 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 $\square G_i$ 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 $a, b \in A(\square G_i)$, $a < b$ if and only if a precedes b on some directed path in $\square G_i$. From the structure it then follows that the ordering of the arcs in the individual G_i is preserved in the Cartesian product, in the following sense. If $a < b$ for two arcs $a, b \in A(\square G_i)$ of the same type G_i, then for the corresponding arcs a' of a in G_i and b' of b in G_i, it holds that $a' < b'$ in G_i. The simplest way to see this is by the level vectors: if $b' < a'$, then the level of the head of b' in G_i is smaller than the level of the tail of a' in G_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 $\square G_i$, contradicting that there is a directed path in $\square G_i$ in which a precedes b.

Remark 3.3.4. In the above proof, we did not specifically consider the possibility of having multiple arcs, and we did not use the labels 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 G_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 $G_1 \square G_2$ corresponds to either an arc in G_1 or an arc in G_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 3.3.2 above).

Remark 3.3.5. 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. We give the general case in Appendix II.

3.4 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 Wöhrle and Thomas (2004). Figure 3.6 gives for our example the transformation of $SQ = MS \square RS \square OD$ consisting of the Cartesian product of the graphs MS, RS, OD to the weak synchronised product $MS \boxminus RS \boxminus OD$.

The weak synchronised product $G_1 \boxminus G_2$ of G_1 and G_2 is defined as the graph on vertex set $V(G_1) \times V(G_2)$ (the Cartesian product of the vertex sets) and arc set $A_{1,2}$ with four types of arcs.

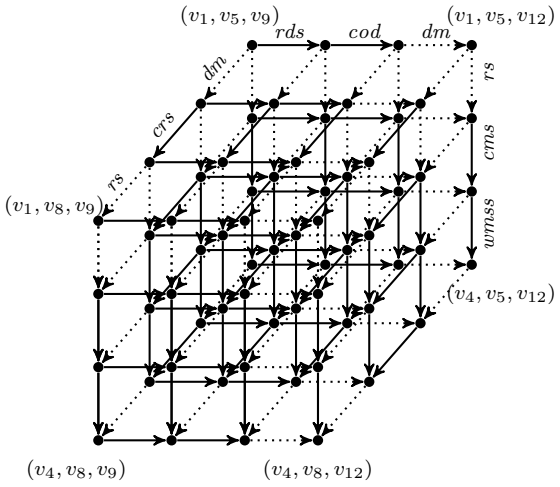


Figure 3.6: Sequence control processes of a mobile robot, from $\square$ to $\boxminus$.

The first two types correspond to arcs in G_1 and G_2 that have labels that only appear in one of G_1 and G_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 $x \in V(G_2)$ and the associated label $\lambda(v_1v_2)$ (*a-type* G_1 arcs) or $(y, w_1)(y, w_2)$ with $y \in V(G_1)$ and the associated label $\lambda(w_1w_2)$ (*a-type* G_2 arcs), where for arcs

$v_1v_2 \in A(G_1)$ label $\lambda(v_1v_2)$ does not appear in G_2 and for arcs $w_1w_2 \in A(G_2)$ label $\lambda(w_1w_2)$ does not appear in G_1.

The other types correspond to arcs in G_1 and G_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 $v_1v_2 \in A(G_1)$ and $w_1w_2 \in A(G_2)$ label $\lambda(v_1v_2) = \lambda(w_1w_2)$. The first and third are *s-type* G_1-arcs and the others are *s-type* G_2-arcs.

For $k \geqslant 3$, the weak synchronised product $G_1 \boxminus G_2 \boxminus \ldots \boxminus G_k$ is defined recursively as $((G_1 \boxminus G_2) \boxminus \ldots) \boxminus G_k$. If no confusion can arise, we denote it as $\boxminus G_i$.

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 3.2 and 3.6, 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 3.4.1. *Let G_i be an acyclic graph for $i = 1, 2, \ldots, k$. Then $\boxminus G_i$ is acyclic and $\ell(\boxminus G_i) \leqslant \ell(\square G_i)$.*

Proof. As in the proof of Lemma 3.3.1, it suffices to prove the statements for $k = 2$, since for integers $k \geqslant 3$, the weak synchronised product $G_1 \boxminus G_2 \boxminus \ldots \boxminus G_k$ is defined recursively as $((G_1 \boxminus G_2) \boxminus \ldots) \boxminus G_k$, hence $G'_1 \boxminus G'_2$, and the result follows by induction.

It is obvious from the definitions that $G_1 \boxminus G_2$ is a spanning subgraph of $G_1 \square G_2$, i.e., that the vertex set of $G_1 \square G_2$ equals the vertex set of $G_1 \boxminus G_2$, and that the arc set of $G_1 \boxminus G_2$ is a subset of the arc set of $G_1 \square G_2$. From this observation, it follows by Lemma 3.3.1 that $G_1 \boxminus G_2$ is acyclic and that $\ell(G_1 \boxminus G_2) \leqslant \ell(G_1 \square G_2)$. □

Remark 3.4.2. It is not difficult to give examples of labelled, directed, acyclic graphs G_1 and G_2 with $\ell(G_1 \boxminus G_2) < \ell(G_1 \square G_2)$.

For example, Figure 3.7 shows G_1 consisting of one directed path $v_1v_2v_3$ with labels $\lambda(v_1v_2) = a$ and $\lambda(v_2v_3) = b$, and G_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 3.8, 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

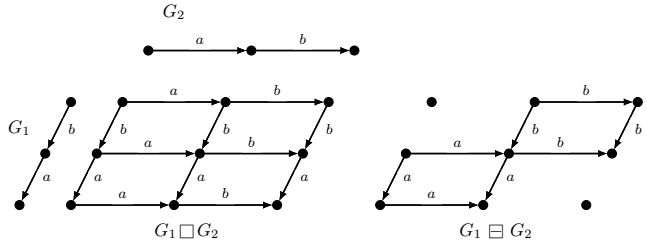


Figure 3.7: Two-dimensional pathological case.

in Figure 3.8 is marked by a circle around the vertex. From three dimensions it is not difficult to extend to n-dimensions.

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

Let P be a longest path in $G_1 \square G_2$. Then P is the interleaved concatenation of two longest paths R and Q in G_1 and G_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 $G_1 \boxminus G_2$, 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 $G_1 \boxminus G_2$ contains a path with the same length as P. So, if $\ell(G_1 \boxminus G_2) < \ell(G_1 \square G_2)$, then 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 G_i are therefore consistent. This then automatically implies that for consistent graphs G_i, $\ell(\boxminus G_i) = \ell(\square G_i)$.

Remark 3.4.3. From the fact that $\boxminus G_i$ is a spanning subgraph of $\square G_i$, it follows that the ordering of the arcs in the individual G_i is preserved in the weak synchronised product, in the following sense. If $a < b$ for two arcs $a, b \in A(\boxminus G_i)$ of a-type or s-type for the same G_i, then for the corresponding arcs a' of a in G_i and b' of b in G_i, it holds that $a' < b'$ in G_i.

This can be seen by using the level vectors. Suppose $b' < a'$, Then the level of the head of b' in G_i is smaller than the level of the tail of a' in G_i. This means

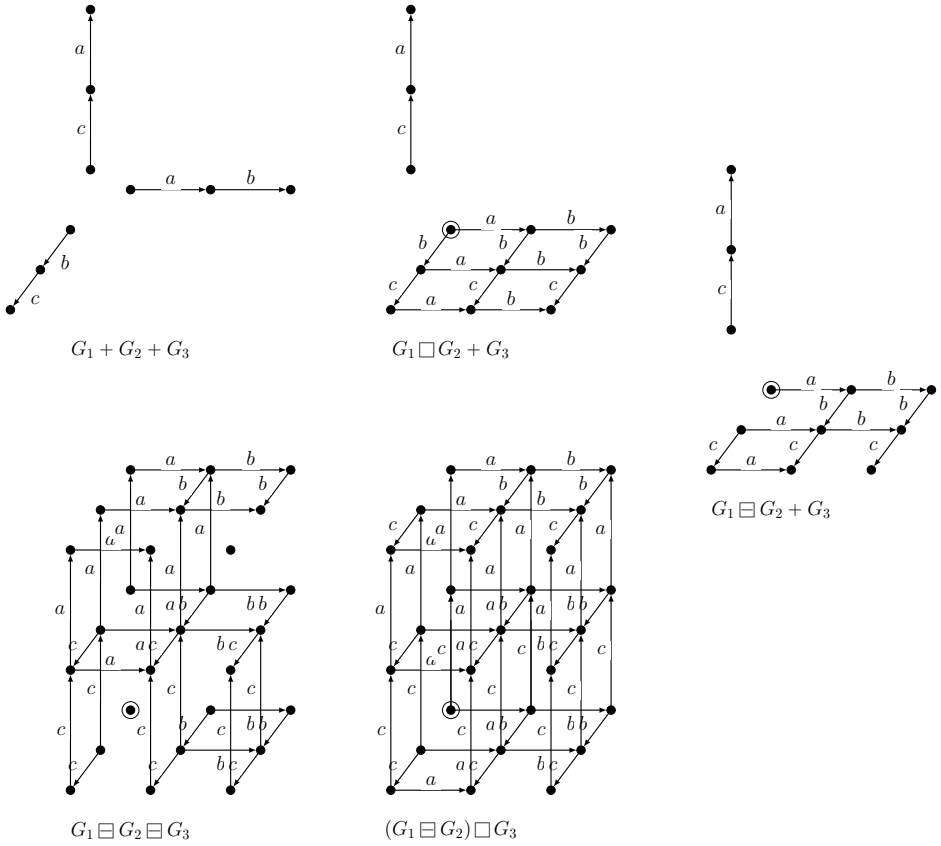


Figure 3.8: Three-dimensional pathological case from $G_1 + G_2 + G_3$, through $G_1 \square G_2 + G_3, G_1 \boxminus G_2 + G_3, (G_1 \boxminus G_2) \square G_3$ to $G_1 \boxminus G_2 \boxminus G_3$.

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 $\boxminus G_i$, contradicting that there is a directed path in $\boxminus G_i$ in which a precedes b. Therefore, the supposition is false.

Remark 3.4.4. 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 3.2 and 3.6 ($\square \Rightarrow \boxminus$). 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.

3.5 The Reduced Weak Synchronised Product of a Set of Parallel Processes

The reduced weak synchronised product $G_1 \boxdot G_2$ of G_1 and G_2 is defined as the graph obtained from the weak synchronised product $G_1 \boxminus G_2$ of G_1 and G_2 by first removing all vertices with level 0 in $G_1 \boxminus G_2$ that have level > 0 in $G_1 \square G_2$, 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 $G_1 \square G_2$. The resulting graph for our standard example is shown in Figure 3.9.

For $k \geqslant 3$, the reduced weak synchronised product $G_1 \boxdot G_2 \boxdot \ldots \boxdot G_k$ is defined recursively as $((G_1 \boxdot G_2) \boxdot \ldots) \boxdot G_k$, and denoted as $\boxdot G_i$ if no confusion can arise.

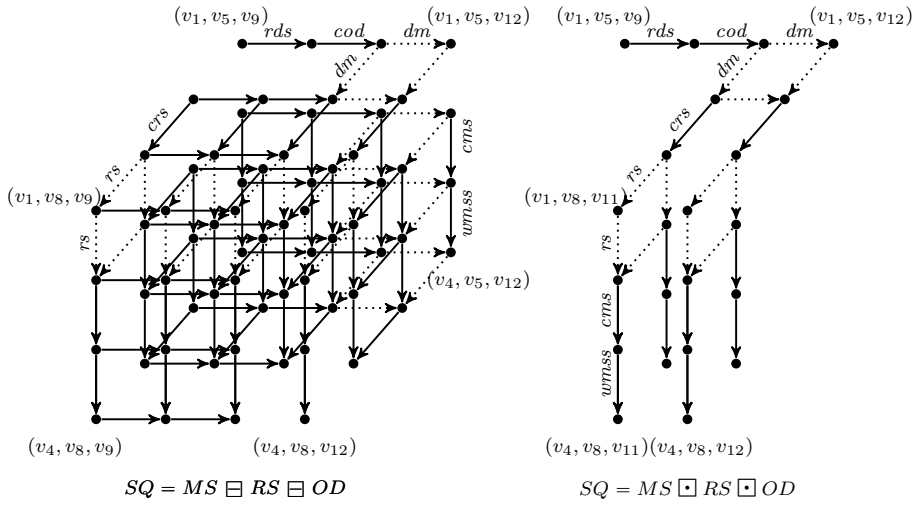


Figure 3.9: Sequence control processes of a mobile robot, from $\boxminus$ to $\boxdot$.

Lemma 3.5.1. *Let G_i be an acyclic graph and let $\boxdot\ G_i$ be the reduced weak synchronised product of G_i for $i = 1, 2, \ldots, k$. Then $\ell(\boxdot G_i) = \ell(\boxminus G_i)$.*

Proof. One direction is clear: since $\boxdot G_i$ is a subgraph of $\boxminus G_i$, we have that $\ell(\boxdot G_i) \leqslant \ell(\boxminus G_i)$. For the other direction, consider a longest path P in $\boxminus G_i$. 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 $\boxminus G_i$, so P is a path and therefore a longest path in $\boxdot G_i$. This completes the proof of Lemma 3.5.1. $\square$

Remark 3.5.2. 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.

Remark 3.5.3. 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 3.9. In Section 3.6, we translate additional restrictions into our product.

3.6 The VRSP of a Set of Parallel Processes

The VRSP of G_1 and G_2, $G_1 \boxtimes G_2$, 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 $G_1 \boxtimes G_2$ are called synchronous arcs, and the set of these arcs is denoted as $A^s_{1,2}$. This intermediate stage is shown in Figure 3.10.

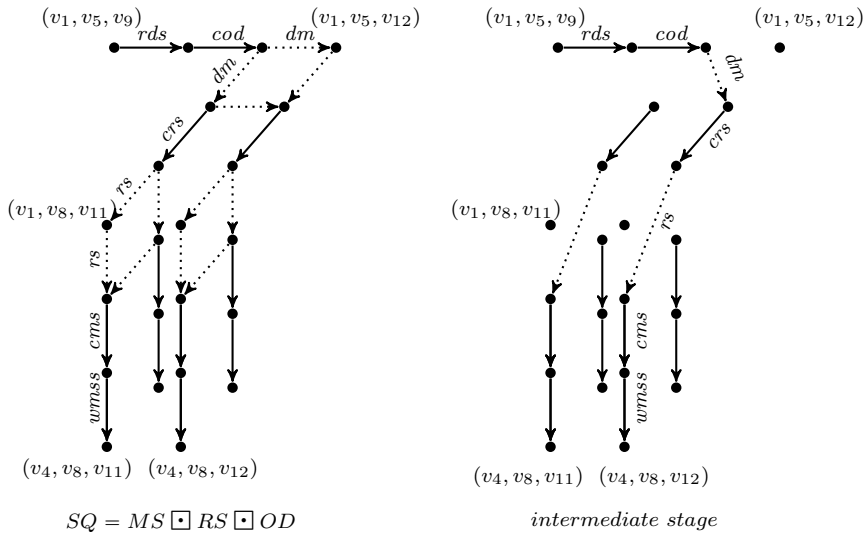


Figure 3.10: Sequence control processes of a mobile robot, from $\boxdot$ to the intermediate stage.

Secondly, all vertices with level 0 in the resulting graph that have level > 0 in $G_1 \square G_2$ 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 $G_1 \square G_2$. The resulting graph is called the VRSP of G_1 and G_2 and denoted as $G_1 \boxbslash G_2$. The set of arcs consisting of the other remaining (asynchronous) arcs of $G_1 \boxbslash G_2$ is denoted as $A^a_{1,2}$.

For $k \geqslant 3$, the VRSP $G_1 \boxbslash G_2 \boxbslash \ldots \boxbslash G_k$ is defined recursively as $((G_1 \boxbslash G_2) \boxbslash \ldots) \boxbslash G_k$, and denoted as $\boxbslash G_i$ if no confusion can arise.

The resulting graph for our standard example is shown in Figure 3.11.

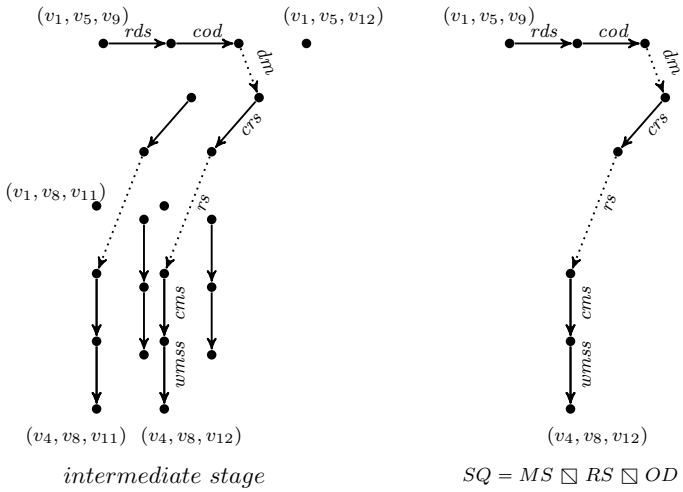


Figure 3.11: Sequence control processes of a mobile robot, from the intermediate stage to $\boxbslash$.

Lemma 3.6.1. *Let G_i be an acyclic graph and let $\boxbslash G_i$ be the VRSP of G_i for $i = 1, 2, \ldots, k$. Then $\ell(\boxbslash G_i) \leqslant \ell(\boxdot G_i)$.*

Proof. As in the proof of Lemma 3.3.1, it suffices to prove the statement for $k = 2$, since for integers $k \geqslant 3$, the VRSP $G_1 \boxbslash G_2 \boxbslash \ldots \boxbslash G_k$ is defined recursively as $((G_1 \boxbslash G_2) \boxbslash \ldots) \boxbslash G_k$, hence $G'_1 \boxbslash G'_2$, and the result follows by induction.

From the definitions of the reduced weak synchronised product and the VRSP, it follows that the vertex set of $G_1 \boxbslash G_2$ is a subset of the vertex set of $G_1 \boxdot G_2$, and the asynchronous arc set $A^a_{1,2}$ of $G_1 \boxbslash G_2$ is a subset of the asynchronous arc set of $G_1 \boxdot G_2$. For the synchronous arc set $A^s_{1,2}$ of $G_1 \boxbslash G_2$ every arc replaces a quadruple of arcs in $G_1 \boxdot G_2$, 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 $\lambda((v_1, w_1)(v_2, w_1))$

$= \lambda((v_1,w_1)(v_1,w_2)) = \lambda((v_1,w_2)(v_2,w_2)) = \lambda((v_2,w_1)(v_2,w_2))$ in $G_1 \boxdot G_2$ 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 VRSP 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 $G_1 \boxdot G_2$ is acyclic, $G_1 \boxtimes G_2$ is acyclic and $\ell(G_1 \boxtimes G_2) \leqslant \ell(G_1 \boxdot G_2)$. □

Remark 3.6.2. The proof of Lemma 3.6.1 shows that combining processes *may* lead to a performance gain, where the gain $\mathcal{G}$ is defined by $\mathcal{G} = \sum_{i=1}^{k} \ell(G_i) - \ell(\boxtimes_{i=1}^{k} G_i)$. It is clear from the above that a gain is only guaranteed if $\ell(\boxtimes G_i) < \ell(\boxdot G_i)$. 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 G_i and for every longest path Q in G_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 VRSP with respect to the length of its constituent graphs. It is obvious (follows from Lemma 3.6.1) 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 3.6.3. *Let G_i be an acyclic graph for $i = 1, 2, \ldots, k$. Then $\ell(\boxtimes G_i) = \ell(G_1) + \ell(G_2) + \ldots + \ell(G_k)$ if and only if every G_i has at least one longest path without synchronising arcs.*

Proof.
Note that it suffices to prove the statement for $k = 2$, since for integers $k \geqslant 3$, $G_1 \boxtimes G_2 \boxtimes \ldots \boxtimes G_k$ is $((G_1 \boxtimes G_2) \boxtimes \ldots) \boxtimes G_k$, hence $G_1' \boxtimes G_2'$, and the result follows by induction.

Firstly, we prove that if $\ell(G_1 \boxtimes G_2) = \ell(G_1) + \ell(G_2)$ then G_1 and G_2 have at least one longest path without synchronising arcs.

The proof is by contraposition. Suppose that all longest paths $P = p_1 p_2 \ldots p_{k_1}, Q = q_1 q_2 \ldots q_{k_2}$ of G_1 and G_2, without loss of generality, contain at least one synchronising arc a with label $\lambda(a)$, say from p_i to p_{i+1} and q_j to q_{j+1}. The VRSP of paths P and Q is $P' \square Q' \cup (p_i p_{i+1} \boxtimes q_j q_{j+1}) \cup P'' \square Q''$, 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 $\ell(P \boxtimes Q) = \ell(P' \square Q') + \ell((p_i p_{i+1} \boxtimes q_j q_{j+1})) + \ell(P'' \square Q'')$.
Note that $p_i p_{i+1}$ and $q_j q_{j+1}$ have the same label and therefore the same weight t. Therefore, $\ell(p_i p_{i+1} \square q_j q_{j+1}) = 2 \times t = 2 \times \ell(p_i p_{i+1} \boxtimes q_j q_{j+1})$ (due to the synchronisation constraint) and it follows that $\ell(P \boxtimes Q) = \ell(P' \square Q') + \ell((p_i p_{i+1} \boxtimes q_j q_{j+1})) + \ell(P'' \square Q'') = \ell(P' \square Q') + t + \ell(P'' \square Q'') < \ell(P' \square Q') + 2 \times t + \ell(P'' \square Q'') =$

$\ell(P'\square Q') + \ell((p_i p_{i+1} \square q_j q_{j+1})) + \ell(P''\square Q'')$, $\ell(P\square Q)$ so the VRSP will reduce the length of the longest paths in G_1 and G_2. This leads to $\ell(G_1\square G_2) > \ell(G_1 \boxtimes G_2)$. Therefore the supposition is false and G_1 and G_2 have at least one longest path without synchronising arcs.

Secondly, we prove that if G_1 and G_2 have at least one longest path without synchronising arcs then $\ell(G_1 \boxtimes G_2) = \ell(G_1) + \ell(G_2)$.

By Lemma 3.3.1 $\ell(G_1\square G_2) = \ell(G_1) + \ell(G_2)$. If $P = p_1 p_2 \dots p_{k_1}$ and $Q = q_1 q_2 \dots q_{k_2}$ are two vertex-disjoint longest paths without synchronising arcs of G_1, G_2 respectively, then clearly $P\square Q$ contains the path PQ, where PQ denotes the path $PQ = (p_1, q_1)(p_1, q_2)\dots(p_1, q_{k_2})(p_2, q_{k_2})\dots(p_{k_1}, q_{k_2})$. By the definition of $G_1 \boxminus G_2$, it follows that $G_1 \boxminus G_2$ contains the path PQ, even so by definition $G_1 \boxdot G_2$ and $G_1 \boxtimes G_2$ contain the path PQ. As $\ell(PQ) = \ell(G_1) + \ell(G_2)$ it follows that $\ell(G_1) + \ell(G_2) = \ell(G_1 \boxtimes G_2)$. This completes the proof of Lemma 3.6.3. □

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

Theorem 3.6.4. *Let G_i be an acyclic graph for $i = 1, 2, \dots, k$. Then $\ell(\boxtimes G_i) < \ell(\boxdot G_i)$ if there exists G_n, G_m, $n \neq m, 1 \leqslant n, m \leqslant k$, such that each longest path in G_n, G_m, contains at least one same labelled synchronising arc.*

Proof. Again it suffices to prove the statements for $k = 2$, since for integers $k \geqslant 3$, $G_1 \boxtimes G_2 \boxtimes \dots \boxtimes G_k$ is $((G_1 \boxtimes G_2) \boxtimes \dots) \boxtimes G_k$, hence $G_1' \boxtimes G_2'$, and the result follows by induction.
From Lemma 3.6.3 we have that every G_i has at least one longest path without synchronising arcs if and only if $\ell(G_1 \boxtimes G_2 \boxtimes \dots \boxtimes G_k) = \ell(G_1) + \ell(G_2) + \dots + \ell(G_k)$, therefore, as both G_1 and G_2 contain only longest paths with at least one synchronising arc, both G_1 and G_2 do not contain a longest path without synchronising arcs. From this observation it follows that $\ell(G_1 \boxtimes G_2) \neq \ell(G_1) + \ell(G_2)$. By Lemma 3.6.1, $\ell(G_1 \boxtimes G_2) \leqslant \ell(G_1 \boxdot G_2)$, it follows that $\ell(G_1 \boxtimes G_2) < \ell(G_1) + \ell(G_2)$. Together with the observation that $\ell(G_1 \boxdot G_2) = \ell(G_1) + \ell(G_2)$ this gives $\ell(G_1 \boxtimes G_2) < \ell(G_1 \boxdot G_2)$. □

Remark 3.6.5. Theorem 3.6.4 seems rather restrictive, but we cannot loosen the requirements on two graphs containing only longest paths with synchronising arcs, to one graph containing only longest paths with synchronising arcs and another graph containing at least one longest path containing a synchronising arc. Of course, this will lead to $\ell(G_1 \boxtimes G_2) < \ell(G_1 \boxdot G_2)$, but the graph $G_1 \boxtimes G_2$ will represent a process with a deadlock, namely a longest path $v_1 v_2 \cdots v_k$ without a synchronising arc in, for example, G_1 and any longest path $w_1 w_2 \cdots w_n$ with a synchronising arc $w_i w_j, j < n$ in G_2, will produce a path in $G_1 \boxtimes G_2$ with a last vertex (v_k, w_i) in stead of (v_k, w_n). In Boode et al. (2013) the erroneous conclusion was that Theorem 1 could be relaxed.

In fact, any path in both graphs must contain synchronising arcs. A formal definition of consistency that solves these problems is given in Definition 6.2.1 on page 73.

3.7 Conclusions

With Theorem 3.6.4, 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 both processes must contain synchronising actions. 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 cases 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 3.6.1 and Remark 3.4.1, 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 not-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 of which the length is reduced when the weak synchronised product is taken.

Finally, we have shown how to get to the VRSP. More importantly, we have shown that potentially we can use the VRSP to improve the worst-case performance of parallel processes. 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 cannot be guaranteed and 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.

4

The VRSP

This chapter is based on our paper presented at the CPA 2014 conference (Boode and Broenink, 2014).

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 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 the mutual exclusion of resources, where a number of processes are 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 chapter. As explained in the introduction, the latter is not a part of this research.

In Boode et al. (2013) we defined periodic real-time processes as finite, labelled, acyclic, directed multigraphs, 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 Boode et al. (2013) a Vertex-Removing Synchronised Product (VRSP) to reduce the number of context switches. The VRSP is based on the synchronised product of Wöhrle and Thomas (2004), 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 (Boode et al., 2013).

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 4.1.

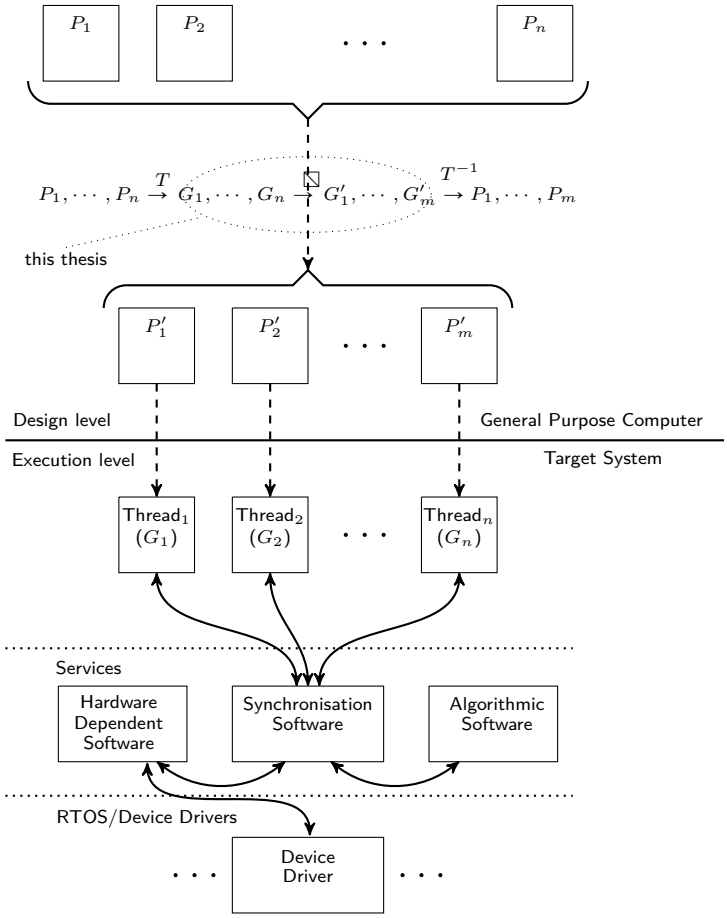


Figure 4.1: Overall System Architecture.

On Design level the designer gives a specification using some process algebra.

Using the 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 process.

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

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 an 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 chapter is an improvement on the design cycle and is illustrated in Figure 4.2.

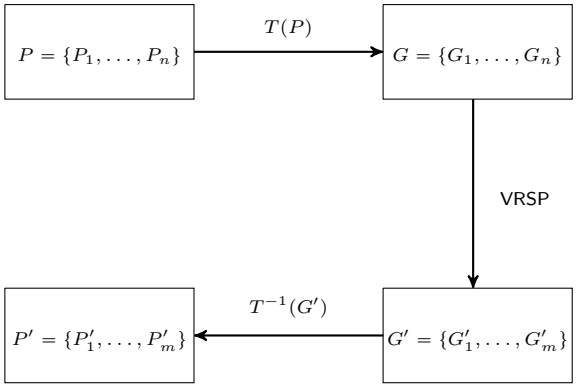


Figure 4.2: The design cycle.

In this cycle we start with the process specification P written in some process-algebraic form. By a transformation function T, $T : P \rightarrow \{G_i\}$, we get a set of finite, labelled, acyclic, directed multigraphs. 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-bisimilar 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 by using the inverse T^{-1} of the transformation function T.

To be able to compose the set of graphs in a meaningful manner, the VRSP has to be idempotent, commutative and associative. We formalise these algebraic properties in Chapter 6. There we show that the VRSP is idempotent and commutative, and we characterise in which cases the VRSP is associative.

Furthermore, we investigate the number of different ways the VRSP can result in a combination of graphs. These products are represented by a lattice. The lattice shows all possible outcomes in case the VRSP is associative. An outcome forms a solution (defined in Section 4.2) for our PHRCS if the processes represented by the

outcome do not miss their deadlines and fit in the available memory. The number of outcomes grows extremely fast in terms of the number of processes. Therefore, we give heuristics for obtaining a solution. These heuristics are not guaranteed to find such a solution, let alone an optimal solution.

An introduction, giving some intuition for the used terminology, is given in Section 4.1. In Section 4.2 we define the notion of solution and outcome for a set of graphs. In Section 4.3 we give a lattice representation of all outcomes of the VRSPs for a set of graphs. In Subsection 4.4.1 through 4.4.3 we give heuristics for calculating a solution for the initial set of graphs. In Section 4.5 we present a case-study where the performance of a model of a Production Cell is optimised. In Section 4.6 we give the conclusions.

4.1 The VRSP

In this section, we give an informal introduction to the VRSP.

In Boode et al. (2013, p. 58) 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 multigraph, 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 4.3).

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.

Processes that have actions in common, will synchronise over these so-called synchronous actions. The VRSP adds to the Cartesian product that whenever there is a quadrangle 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 4.4.)

The two processes represented by the components G_1 and G_2 contain two and three actions. To execute the processes represented by these components, there will be five context switches. By using the VRSP to create the process represented by $G_1 \boxtimes G_2$, 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[2] is part of

[2]Roscoe (1998) defines the tock-action as "the drum-beat of the passage of time as an explicit event."

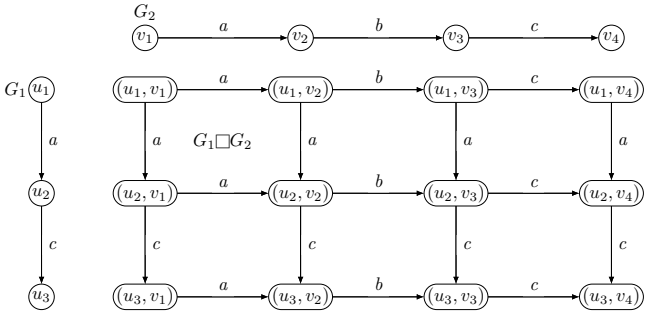


Figure 4.3: The Cartesian product $G_1 \square G_2$ of G_1 and G_2.

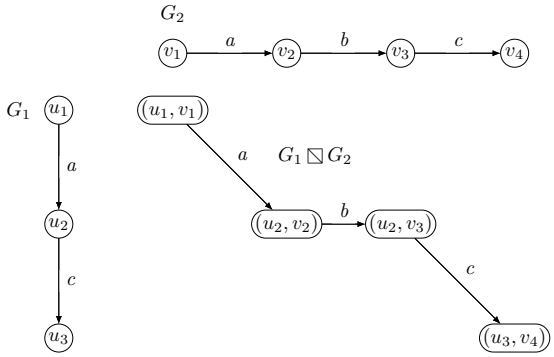


Figure 4.4: The VRSP $G_1 \boxtimes G_2$ of G_1 and G_2.

every process and will be part of synchronisation. We use $\ell(G_i)$ to denote the maximum length of a path in a component. This represents 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 length of a longest path of the graph is then a summation over the length of a longest path of each component of the graph, $\sum_i \ell(G_i)$.

In general, the processes represented by these components may suffer from deadlocks. Two processes, represented by components G_1 and G_2, are *consistent* if the following holds. For every path P of G_1 there exists a subgraph P' with an arc set corresponding to the arcs of P (one arc for each arc of P, so P' is a set of paths) such that P' is contained in a path Q of $G_1 \boxtimes G_2$ (so Q may contain arcs corresponding to arcs of G_2). We require an analogous property for paths of G_2. Additionally, we require that the source and the sink of $G_1 \boxtimes G_2$ are the Cartesian product of the sources and sinks of G_1 and G_2, respectively. The formal definition of consistency of graphs will be given in Definition 6.2.1 on page 73.

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 a length greater than 2, the system will be pairwise consistent. If we reduce such a deadlock cycle down to just 2 processes through the VRSP, those two processes will not be consistent - exposing the deadlock. So, without deadlock freedom, the VRSP does not preserve pairwise consistency - see Figure 4.5. In fact, this shows that we have to check the consistency whenever we apply the VRSP.

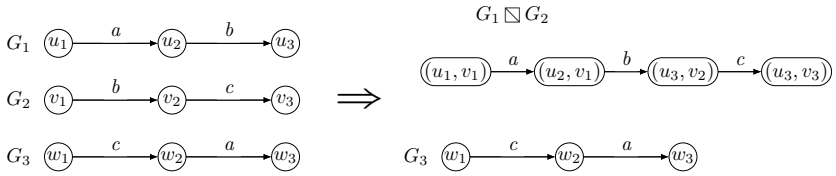


Figure 4.5: The VRSP is not preserving pairwise consistency.

The first such check will take n^2 operations. 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 will take only n operations. If we continue until a single process remains without any pairwise consistency check failing, the system must have been deadlock free.

However, we also need pairwise consistency for the VRSP to be associative (see Figure 4.6). Without associativity the behaviour of the resulting process(es) corresponding to the VRSP of the associated graphs would depend on the order in which we compute the VRSP.

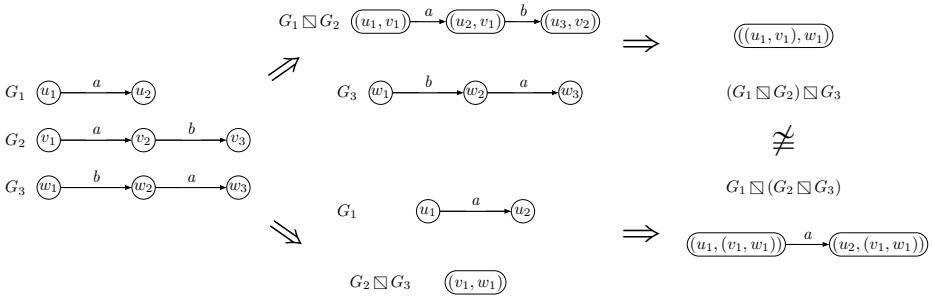


Figure 4.6: Non-associativity of not pairwise consistent components.

4.2 The Solution Set for the VRSP

So far, we have mentioned that we seek a solution for our performance problem of reducing the number of context-switches for a PHRCS, but we did not specify what a solution is for a set of graphs.

If we start with two graphs G_1 and G_2, there are two options: we take the disjoint union $G_1 + G_2$ or we apply the (commutative) VRSP and take $G_1 \diagdown\!\!\diagdown G_2$. If we start with three graphs G_1, G_2 and G_3, we have quite a few more options: apart from taking $G_1 + G_2 + G_3$, $G_1 + (G_2 \diagdown\!\!\diagdown G_3)$, $G_2 + (G_1 \diagdown\!\!\diagdown G_3)$ and $G_3 + (G_1 \diagdown\!\!\diagdown G_2)$, we can take the VRSP of all three of the graphs; in the latter case, we have one or three different options, depending on whether the VRSP is associative or not. No matter whether the VRSP is associative or not, we will use the term (different) *outcome* for each of the (different) options. We let $N_{\omega(G)}$ denote the number of different outcomes when we apply the VRSP to the graph $G = \Sigma_{i=1}^{\omega(G)} G_i$. In case the components of G are chosen in such a way that the VRSP is guaranteed to be associative, we indicate this by using $N^a_{\omega(G)}$ instead of $N_{\omega(G)}$.

We say an outcome is a *solution* if the processes they represent meet the requirements with respect to the deadlines and the memory occupancy.

4.3 The VRSP as a Lattice

By applying the the VRSP in order to produce a solution for our optimisation problem, in the worst case we have to calculate all outcomes. We are going to show in Chapter 5 that the number of outcomes is given by the Bell number in case the VRSP is associative, and by the Bessel number in case the VRSP is non-associative. In the remainder of this chapter we only deal with the associative case.

Using the binary operations + (the disjoint union of graphs) and $\diagdown\!\!\diagdown$ (the VRSP of graphs) we can depict all the possible outcomes in a lattice (Birkhoff, 1984). As shown in Figure 4.7, the top node of this lattice represents $\sum_{i=1}^{n} G_i = G_1 + G_2 + \ldots + G_n$ and the bottom node of this lattice represents $\bigdiagdown_{i=1}^{n} G_i$. Furthermore, two nodes in the lattice are adjacent (from top to bottom) if in the graph represented by the upper node, two components are multiplied by the VRSP, leading to a disjoint union of components represented by the lower node. As an example, there are exactly three ways to produce $G_1 \diagdown\!\!\diagdown G_3 \diagdown\!\!\diagdown G_4 + G_2$ from $G_1 + G_2 + G_3 + G_4$: by first applying the VRSP to G_1 and G_3, and then applying the VRSP to $G_1 \diagdown\!\!\diagdown G_3$ and G_4; by first applying the VRSP to G_1 and G_4, and then applying the VRSP to $G_1 \diagdown\!\!\diagdown G_4$ and G_3; and finally, by applying the VRSP to G_3 and G_4, and then applying the VRSP to $G_3 \diagdown\!\!\diagdown G_4$ and G_1. These three ways are depicted by the thick lines in Figure 4.7.

The first position in the index of a node in Figure 4.7 is related to G_1, the second to G_2, and so on. All zeros in the index stand for the disjoint union of all the

graphs related to the positions of the zeros. All integers $i \neq 0$ stand for the VRSP of all the graphs related to the positions of the is. Note that different integers in the index denote disjoint unions of the VRSPs corresponding to these integers. For example, the node v_{10122} denotes $G_1 \boxtimes G_3 + G_2 + G_4 \boxtimes G_5$.

The nodes in the lattice represent all possible outcomes when we apply $+$ and $\boxtimes$.

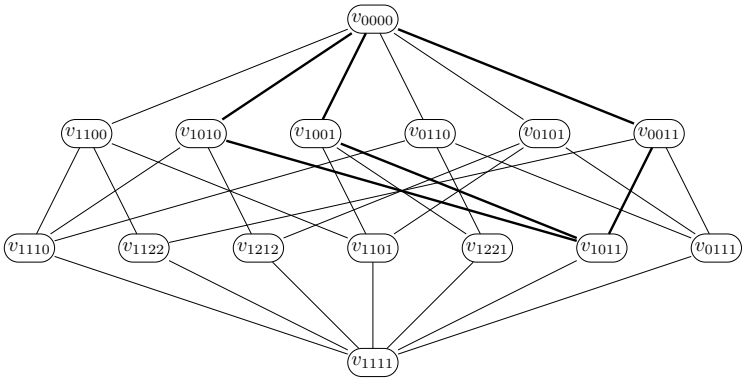


Figure 4.7: The lattice for G_1 to G_4. In bold the possible paths from v_{0000} to v_{1011}.

For a graph $G = \sum_{i=1}^{\omega(G)} G_i$, the depth of the lattice is $\omega(G) - 1$. A node represents a solution G if $\ell(G) \leqslant \mathcal{D}$ and $size(G) \leqslant \mathcal{M}$, where $\mathcal{D}$ is the deadline of the application, $size(G)$ is the amount of memory needed to store the data structure representing the graph G and $\mathcal{M}$ is the available memory to store the data structure representing G.

4.4 Algorithms

Periodic real-time processes are defined as components of a finite, labelled, acyclic, directed multigraph. A longest path in such a graph represents 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 outcomes 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 $n > 20$). 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 lattice generated by all outcomes of the VRSP for a graph G. We describe three heuristics for obtaining a solution in Section 4.4.1 through 4.4.3. Appendix III gives the various pseudocodes.

4.4.1 The Largest Alphabetical Intersection

A simple algorithm 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 $\ell(G_i \boxtimes G_j)$ of G_i and G_j may be equal to the length of the sum $\ell(G_i + G_j)$ of G_i and G_j. Because we do not require that every longest path in G_i synchronises over some action with a full path in G_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 4.8, although the common label set is of size n, the length of the components is reduced by only one.

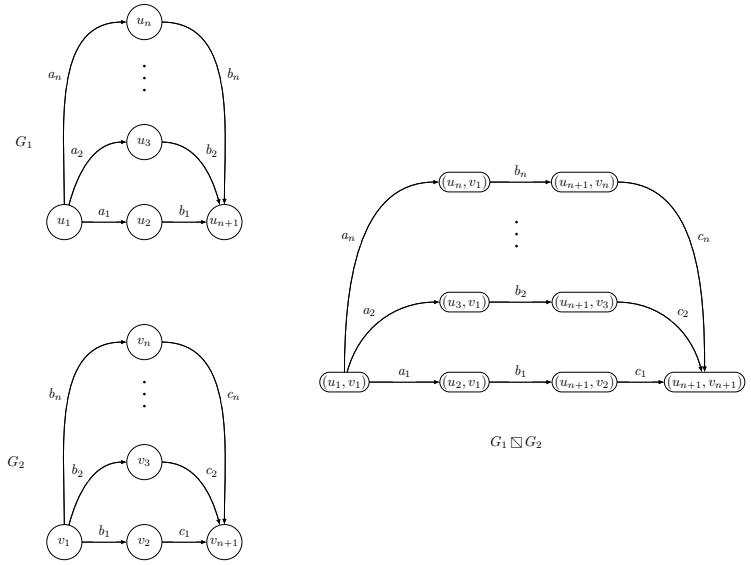


Figure 4.8: Synchronising over choice.

It could even be that the product of two components, due to state-space explosion, is not calculable. LAI is given in Appendix III, Algorithm 5.

4.4.2 Maximising Synchronising Arcs

An adaptation of the algorithm in Section 4.4.1 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.

4.4.3 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 G_1 and G_2 that synchronise over arcs that are at the beginning (G_1) and arcs that are at the end (G_2), may have a very large asynchronous arc set, but the $G_1 \boxtimes G_2$ is linear with respect to the size of $G_1 + G_2$. Without stating the algorithm we have that the selected G_i and G_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.

4.5 The Production Cell Case Study

As a case study we use a Production Cell given in Figure 4.9 (Groothuis et al., 2009). This Production Cell has six optical sensors and six motors. Each motor also contains an angle sensor. For the control loop, the duty cycle is 1 ms.

Veldhuijzen (2009) 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 QNX 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 out-flowing 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 an FSP [3] model of the concurrent processes in Section 4.5.1, followed by a description of the processes in Section 4.5.2. The impact and an example of the synchronised product are discussed in Section 4.5.3. In Section 4.5.4 we analyse the performance data and show the time and space-related behaviour of the presented algorithms. In Section 4.5.5 we discuss the results so far.

[3] 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 (Hoare, 1978).

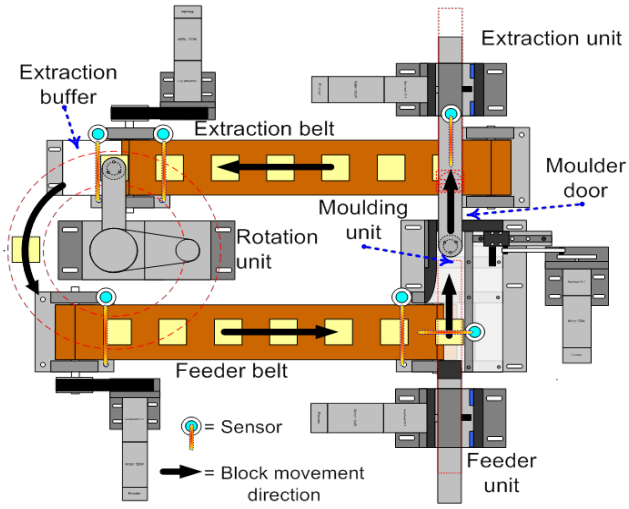


Figure 4.9: Production Cell.

4.5.1 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 4.1. 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%.

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

Listing 4.1: Concurrent Processes of the *Production Cell*.

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 nodes in the lattice.

4.5.2 Process Description

In Listing 4.2, 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 4.10.

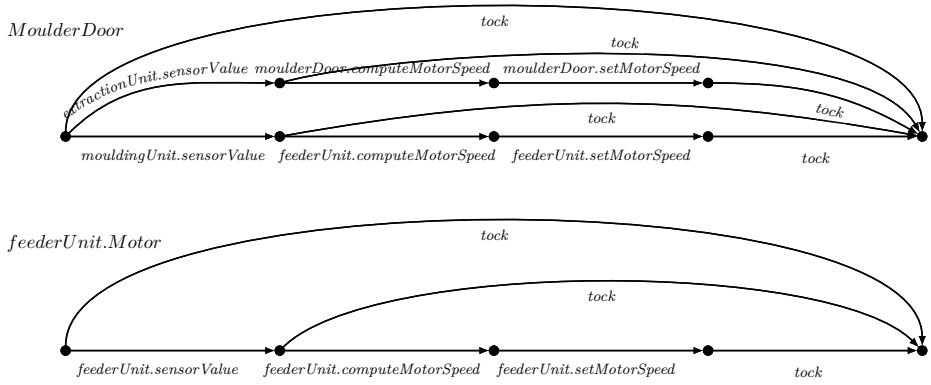


Figure 4.10: Components representing the parallel processes MoulderDoor and feederUnit.Motor.

4.5.3 The VRSPs of the Production Cell

The synchronised product of the processes MoulderDoor and feederUnit.Motor is given in Figure 4.11. It shows a reduction of the longest path length of three. This means that by taking this product, there are three fewer context switches. The memory occupancy is extended by seven units (Appendix IV, Table 2).

Other synchronised products show a reduction of the longest path length (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[4].

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

```
Motor        = (sensorValue → (computeMotorSpeed → setMotorSpeed → tock → MotorStop
               |tock → MotorStop)
               |tock → MotorStop),
MotorStop    = STOP.

Sensor       = (readSensor → calculateSensorValue → (sensorValue → tock → SensorStop
                                                    |tock → SensorStop)
               |tock → SensorStop),
SensorStop   = STOP.

Magnet       = (sensorValue → (angleZero → contraction → tock → MagnetStop
                             |anglePI → release → tock → MagnetStop
                             |tock → MagnetStop)
               |tock → MagnetStop),
MagnetStop   = STOP.

MoulderDoor  = (mouldingUnit.sensorValue →  (feederUnit.computeMotorSpeed
                                             → feederUnit.setMotorSpeed → tock
                                             → MoulderDoorStop
                                            |tock → MoulderDoorStop)
               |extractionUnit.sensorValue → (moulderDoor.computeMotorSpeed
                                             → moulderDoor.setMotorSpeed → tock
                                             → MoulderDoorStop
                                            |tock → MoulderDoorStop)
               |tock → MoulderDoorStop),
MoulderDoorStop = STOP.

Clock = (oneMilliSecondTimer → tock → STOP).
```

Listing 4.2: Description of the *Production Cell*.

4.5.4 Performance of the Production Cell

In Table 4.1 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 $\sum_i \boxtimes_j G_{i,j}$. The memory needed in practice depends on the kind of data structures that will be used for the implementation of the specification. The longest path, $\ell(G_i)$, reflects the maximum number of action related context switches for each process.

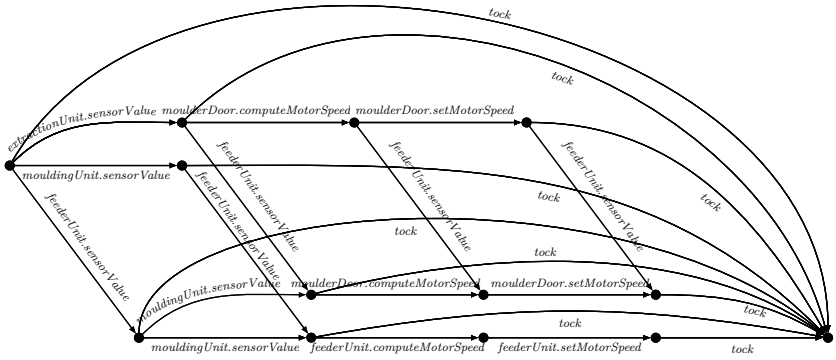


Figure 4.11: The Synchronised Product of the components MoulderDoor and feederUnit.Motor.

i	$Process_i$	$\ell(G_i)$	M		i	$Process_i$	$\ell(G_i)$	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	$angleRotationUnit.Magnet$	4	12
8	$Clock$	2	5		16	$extractionUnit.Magnet$	4	12

Table 4.1: 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 IV, Table 2.

Based on Table 2, Figure 4.12 describes the behaviour of the three algorithms with respect to (the hypothetical values) M and D. The abscissa represents the length of the graph G. 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 represented by M in Figure 4.12 and $\mathcal{D}$ is the deadline for every period represented by D in Figure 4.12. 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 actions. The dotted ellipse shows the component compositions that fulfil the requirements.

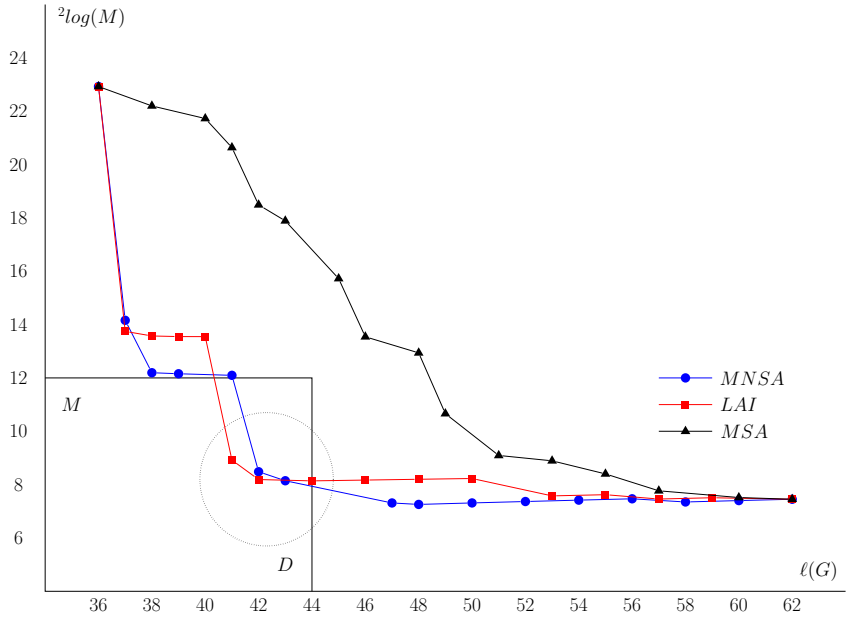


Figure 4.12: Performance of MNSA, LAI, MSA.

Figure 4.12 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 lattice, its synchronised product with repeatedly the other components) component will always be chosen for synchronisation with remaining components. Figure 4.12 shows that the reduction of the $\ell(G)$ leads to a state space explosion from the fifth synchronised product onwards ($\ell(G) = 47$, $^2 log(M) \approx 10.7$).

Of course, it depends on the requirements of the application which vertex in the lattice 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.

4.5.5 Discussion

In practice, the number of parallel processes, and therefore the number of components of the graph G, is often limited to 15 to 20 processes. Our simple case study comprises 16 processes, but complex CPSs may have 50 or more processes. For 15 processes, there are $B(15) \approx 10^9$ nodes in the related lattice; 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 an optimal solution out of all outcomes for 20 processes (assuming the algorithm that calculates an 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.

4.6 Conclusions

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

We have used a lattice to show all possible combinations of additions of products of components. The size of the lattice is exponential with respect to the number of components and is given by the Bell number.

In practice, the number of parallel processes, and therefore the number of components of the graph G, is often limited to 15 or 20 processes, but complex CPSs may have 50 or more processes. With the technology of today, an upper limit of 20 processes is probably still tractable, whereas for complex CPSs of 50 processes or more a heuristic like the ones we proposed must be used.

Clearly, for applications containing hundreds of processes heuristics like MNSA or LAI will give an educated guess which outcomes have to be calculated. Calculations using these heuristics show a performance improvement at the cost of an increasing memory occupancy. 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 approach will only work for the initial set of processes due to the extensive calculations that are necessary.

5

The Number of Outcomes when Applying the VRSP

In the previous chapters, we showed how the performance of a PHRCS might be improved by the use of the VRSP, in particular, if the set of parallel processes of the PHRCS does not meet the requirements regarding timeliness or memory occupancy. We showed as well that, while applying the VRSP to a set of graphs, there are many different outcomes.

In this chapter, we give an expression for the number of possible different outcomes, when applying the VRSP to a set of graphs. We start in Section 5.1 by presenting some additional terminology on trees and forests that we need for our analysis. In Section 5.2, we deal with the associative case. We show that in this case the number of possible outcomes is equal to the Bell number. In Section 5.3, we deal with the non-associative case. We show that in this case the number of possible outcomes is equal to the Bessel number. We finish in Section 5.4 with the conclusions containing a numerical comparison of the non-associative case and the associative case.

5.1 Terminology of Trees and Forests

We use Bondy and Murty (2008) for terminology and notations on graphs not defined here.

Let T be a *tree*, so a connected acyclic (undirected) graph. We orient the tree by replacing each of the edges of T by an arc, in precisely one of the two directions, so we obtain an acyclic weakly connected directed graph, which we call a *ditree*. A *source* in a ditree is a vertex with in-degree 0. This is usually referred to as a *leaf*. A *sink* in a ditree is a vertex with out-degree 0. We call such a vertex a *target* of the ditree. We say that a ditree D is a *target tree* if D has the following properties. Each vertex except for the leaves has in-degree 2; each vertex except for one has out-degree 1; the unique vertex of D (if D has more than one vertex) with in-degree 2 and out-degree 0 is called the target of D.

The target v of a target tree D will be interpreted as the VRSP of two graphs that are represented by the two in-neighbours u and w of v in D. If u is a target

vertex of $D - v$, then analogously u can be interpreted as the VRSP of two graphs, etc. On the other hand, each of the ways to compute the product of the graphs $G_1, \ldots, G_n$ can be represented as a target tree on n leaves and $n-1$ internal vertices (non-leaves).

A forest is a graph that consists of one or more ditrees.

5.2 The Associative Case

Let G be a graph with components $G_1, \ldots, G_n$. To compute the number of different outcomes $N^a_{\omega(G)}$ we consider a set $\mathsf{S} = \{G_1, \ldots, G_n\}$. In a partition of S we interpret each element of the partition as a multiplication by the VRSP of the components in that element. The number of partitions of the set S is given by the Bell number B_n with $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k, B_0 = 1$; a proof is given in Cohn et al. (1962). We illustrated this for $n = 4$ with the lattice described in Figure 4.7 on page 50.

5.3 The Non-Associative Case

Counting the number of different outcomes $N_{\omega(G)}$ for the VRSP of a graph G if the VRSP is not associative is not straightforward. We are going to prove that $N_{\omega(G)}$ is equal to the Bessel number $\tilde{B}_{\omega(G)}$ by showing that $N_{\omega(G)}$ satisfies the recurrence relation of the Bessel numbers.

Each outcome can be represented by a forest of binary ditrees. To determine $N_{\omega(G)}$ it is sufficient to determine the number of different forests consisting of ditrees with $\omega(G)$ leaves in total. We give an example in Figure 5.1 showing all 37 forests with in total four leaves.

To determine $N_{\omega(G)}$, we can apply a result due to Erdős (1993). Erdős (1993) defines a semi-labelled tree as a rooted tree with labelled leaves and a semi-labelled forest F as a forest consisting of semi-labelled trees. By taking the G_i as the labels of the leaves, these semi-labelled trees are the same trees as our ditrees. Furthermore, Erdős (1993) uses the double factorial. We adopt Erdős' definition that $(-1)!! = 1$ and that $k!! = k(k-2)(k-4)\cdots 1$ for all odd positive integers k.

Let $b(n, i)$ denotes the number of binary semi-labelled forests with n leaves and i ditrees. Due to Erdős (1993) we have the following expression for $b(n, i)$.

$$b(n,i) = \binom{2n-i-1}{i-1} (2n-2i-1)!!$$

Let $\mathcal{F}_n$ be the number of forests with n leaves distributed over $1, \ldots, n$ ditrees. Then $\mathcal{F}_n = \sum_{i=1}^{n} b(n,i)$, $n \geqslant 1$. It is sufficient to prove that $\mathcal{F}_n = \tilde{B}_n$, $n \geqslant 1$. We do this in Theorem 5.3.4 by showing that $\mathcal{F}_n$ satisfies the recurrence relation of the Bessel numbers.

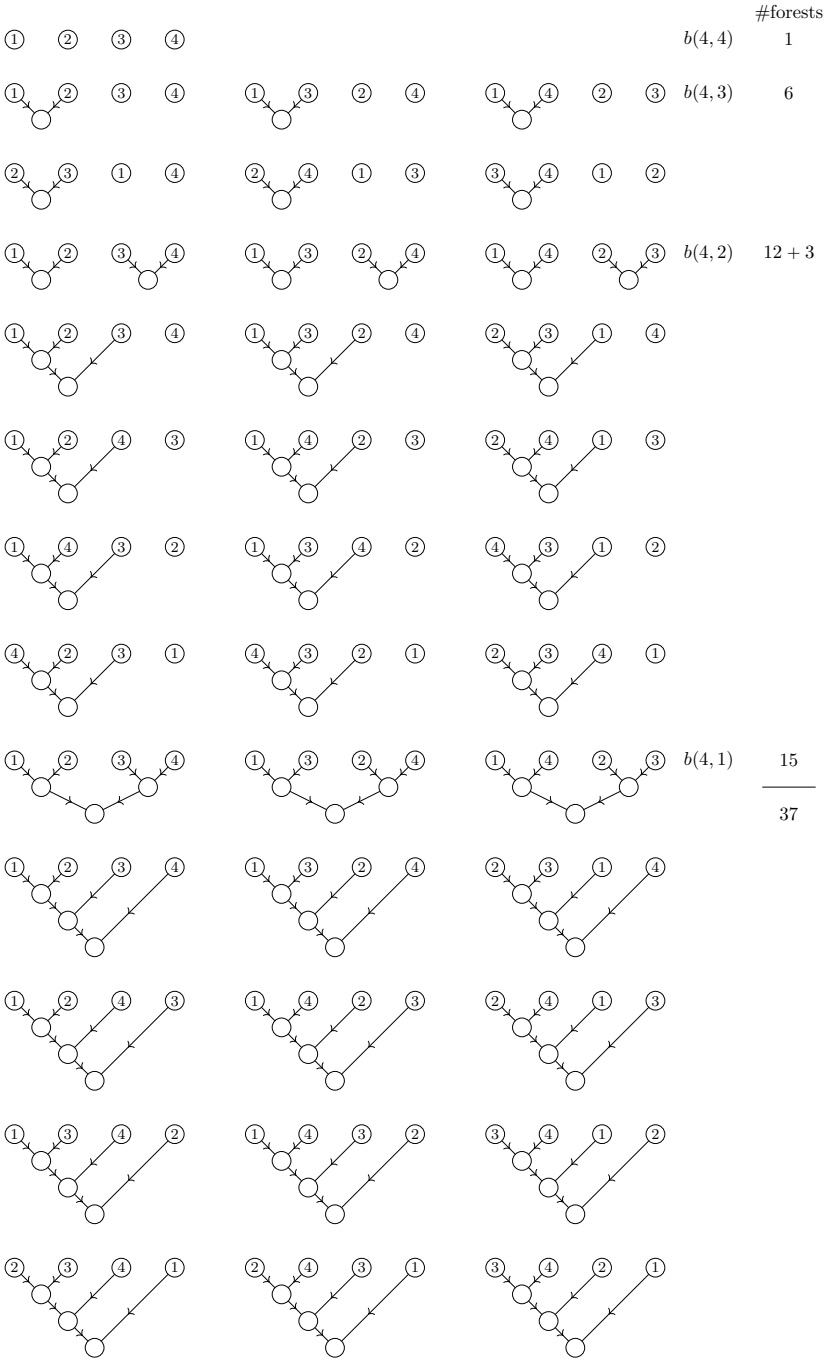


Figure 5.1: Number of forests of binary ditrees for $n = 4$.

First, we prove $b(n+1,1) = (2n-1)b(n,1)$, $n \geqslant 1$, $b(n+1,2) = (2n-1)b(n,2)$, $n \geqslant 2$, and $b(n+1,i) = (2n-1)b(n,i) + b(n-1,i-2)$, $n \geqslant i \geqslant 3$. Then we prove that $\mathcal{F}_n = \tilde{B}_n$, $n \geqslant 0$.

Lemma 5.3.1. *$b(n+1,1) = (2n-1)b(n,1)$, $n \geqslant 1$.*

Proof. To prove the equality, it is sufficient to prove that

$$\begin{aligned}
b(n+1,1) &= (2n-1)b(n,1) && \textit{or}\\
\binom{2n}{0}(2n-1)!! &= (2n-1)\binom{2n-2}{0}(2n-3)!! && \textit{or}\\
(2n-1)!! &= (2n-1)(2n-3)!! && \textit{or}\\
(2n-1)!! &= (2n-1)!!
\end{aligned}$$

The final equality indeed holds, completing the proof of Lemma 5.3.1. □

Lemma 5.3.2. *$b(n+1,2) = (2n-1)b(n,2)$, $n \geqslant 2$.*

Proof. To prove the equality, it is sufficient to prove that

$$\begin{aligned}
b(n+1,2) &= (2n-1)b(n,2) && \textit{or}\\
\binom{2n-1}{1}(2n-3)!! &= (2n-1)\binom{2n-3}{1}(2n-5)!! && \textit{or}\\
(2n-1)(2n-3)!! &= (2n-1)(2n-3)(2n-5)!! && \textit{or}\\
(2n-1)!! &= (2n-1)!!
\end{aligned}$$

The final equality indeed holds, completing the proof of Lemma 5.3.2. □

Lemma 5.3.3. *$b(n+1,i) = (2n-1)b(n,i) + b(n-1,i-2)$, $n \geqslant i \geqslant 3$.*

Proof. To prove $b(n+1,i) = (2n-1)b(n,i) + b(n-1,i-2)$, $n \geqslant i \geqslant 3$, it is sufficient to prove that

$$\begin{aligned}
\binom{2n-i+1}{i-1}(2n-2i+1)!! = {} & (2n-1)\binom{2n-i-1}{i-1}(2n-2i-1)!! \quad + \\
& \binom{2n-i-1}{i-3}(2n-2i+1)!!
\end{aligned}$$

This is equivalent to showing:

$$\begin{aligned}
&\frac{(2n-2i+1)(2n-i+1)(2n-i)}{(2n-2i+2)(2n-2i+1)}\binom{2n-i-1}{i-1}(2n-2i-1)!! = \\
&(2n-1)\binom{2n-i-1}{i-1}(2n-2i-1)!! + \\
&\frac{(i-1)(i-2)(2n-2i+1)}{(2n-2i+2)(2n-2i+1)}\binom{2n-i-1}{i-1}(2n-2i-1)!! \quad \textit{or}
\end{aligned}$$

$$\frac{(2n-i+1)(2n-i)}{(2n-2i+2)} = (2n-1) + \frac{(i-1)(i-2)}{(2n-2i+2)} \qquad or$$

$$(2n-i+1)(2n-i) = (2n-1)(2n-2i+2) + (i-1)(i-2) \qquad or$$

$$4n^2 - 4ni + i^2 + 2n - i = 4n^2 + 2n - 4ni - i + i^2$$

The final equality indeed holds, completing the proof of Lemma 5.3.3. □

To complete this section, we are now going to show that $\mathcal{F}_n$ satisfies the recurrence relation of the Bessel number $\tilde{B}_n$.

Theorem 5.3.4. *$\mathcal{F}_{n+1} = (2n-1)\mathcal{F}_n + \mathcal{F}_{n-1}$, $n \geqslant 2$, $\mathcal{F}_1 = 1$, $\mathcal{F}_2 = 2$.*

Proof. Firstly, we have $\mathcal{F}_n = \sum_{i=1}^{n} b(n,i)$. Next, it is clear that $\mathcal{F}_1 = 1$, $\mathcal{F}_2 = 2$, $b(n,-1) = 0$, and $b(n,n+1) = 0$, $n \geqslant 1$. Using Lemmas 5.3.1, 5.3.2 and 5.3.3 we obtain:

$$\begin{aligned}
b(n+1,1) &= (2n-1)b(n,1) + b(n-1,-1)\\
b(n+1,2) &= (2n-1)b(n,2) + b(n-1,0)\\
&\vdots\\
b(n+1,n) &= (2n-1)b(n,n) + b(n-1,n-2)\\
b(n+1,n+1) &= (2n-1)b(n,n+1) + b(n-1,n-1)
\end{aligned}$$

and it follows that

$$\mathcal{F}_{n+1} = \sum_{i=1}^{n+1} b(n+1,i) = (2n-1)\sum_{i=1}^{n+1} b(n,i) + \sum_{i=1}^{n+1} b(n-1,i-2) = (2n-1)\mathcal{F}_n + \mathcal{F}_{n-1}$$

This completes the proof of Theorem 5.3.4. □

Theorem 5.3.4 shows that $\mathcal{F}_n = \tilde{B}_n$, $n \geqslant 0$.

5.4 Conclusions

The consistency requirement assures that the VRSP is associative and therefore the number of (different) outcomes $N^a_{\omega(G)}$ is equal to the Bell number B_n. This is a vast improvement with respect to the Bessel number series, $\tilde{B}_n$.

Hence, the consistency requirement ensures that not only deadlocks are avoided, but also that the set of possible (different) outcomes is, although rapidly growing, an order of magnitude smaller than the set of possible (different) outcomes without associativity.

In Table 5.1 we show the expansion of B_n and $\tilde{B}_n$ for relatively small n.

n	0	1	2	3	4	5	6	7	8	9	10	11
B_n	1	1	2	5	15	52	203	877	4140	21147	115975	678570
$\tilde{B}_n$	1	1	2	7	37	266	2431	27007	353522	5329837	90960751	1733584106

n	12	13	14	15	16
B_n	4213597	27644437	190899322	1382958545	10480142147
$\tilde{B}_n$	36496226977	841146804577	21065166341402	569600638022431	16539483668991901

Table 5.1: Expansion of B_n and $\tilde{B}_n$.

6

Consistency of Processes and Graphs

As we have already mentioned in Chapter 1, a PHRCS must behave as envisioned by the designer and comply with the established safety requirements. An important aspect of this behaviour is that the PHRCPs must be consistent in the sense that every process must reach a predefined final state before the end of each period. Since we use graphs to represent and deal with these processes, it is natural to try to determine the counterpart of consistency of processes in terms of the associated graphs. In this chapter, we will introduce a notion of consistency of graphs. We will show that the consistency of graphs is equivalent to the consistency of the associated processes.

We start with some additional definitions and terminology in Section 6.1. In Section 6.2, we define the notion of consistency of graphs under the VRSP and analyse its characteristics. In Section 6.3 we formalise the concept of consistency of processes. We show that this consistency concept of processes is equivalent with the consistency of the associated graphs. In Section 6.4, we discuss relevant algebraic properties of the VRSP. We conclude this chapter with some additional discussion and remarks in Section 6.5.

6.1 Terminology

We use Bondy and Murty (2008), Hell and Nešetřil (2004) and Hammack et al. (2011) for terminology and notations on graphs not defined here.

6.1.1 Graph Basics

In the previous chapters, the graphs we considered consisted of a vertex set, an arc set, a set of label pairs, and a mapping assigning the label pairs to the arcs. Although we allowed multiple arcs, we did not treat them in a mathematically very rigorous way. In order to meet our needs with respect to multiple arcs, we adapt the definition of a graph slightly by adding an incidence function, in the following way.

Throughout this chapter, unless we specify explicitly that we consider other types of graphs, all graphs we consider are *finite, deterministic, labelled, acyclic, directed multigraphs*, i.e., they may have multiple arcs. Such graphs consist of a *vertex set*

V (representing the states of a process), an *arc set* A (representing the actions, i.e., transitions from one state to another), a set of *labels* L (in our applications in fact a set of label pairs, each representing a type of action and the worst-case duration of its execution), and two mappings. The first mapping $\mu : A \to V \times V$ is an incidence function that identifies the *tail* and *head* of each arc $a \in A$. In particular, $\mu(a) = (u, v)$ means that the arc a is directed from $u \in V$ to $v \in V$, where $tail(a) = u$ and $head(a) = v$. We also call u and v the *ends* of a. The second mapping $\lambda : A \to L$ assigns a label pair $\lambda(a) = (\ell(a), t(a))$ to each arc $a \in A$, where $\ell(a)$ is a string representing the (name of an) action and $t(a)$ is the *weight* of the arc a. This weight $t(a)$ is a real positive number representing the worst-case execution time of the action represented by $\ell(a)$.

Let G denote a graph according to the above definition. We sometimes use $V(G)$ instead of V, $A(G)$ instead of A, etcetera, in order to avoid confusion. An arc $a \in A(G)$ is called an *in-arc* of $v \in V(G)$ if $head(a) = v$, and an *out-arc* of v if $tail(a) = v$. The *in-degree* of v, denoted by $d^-(v)$, is the number of in-arcs of v in G; the *out-degree* of v, denoted by $d^+(v)$, is the number of out-arcs of v in G. The subset of $V(G)$ consisting of vertices v with $d^-(v) = 0$ is called the *source* of G, and is denoted by $S'(G)$. The subset of $V(G)$ consisting of vertices v with $d^+(v) = 0$ is called the *sink* of G, and is denoted by $S''(G)$.

For disjoint nonempty sets $X, Y \subseteq V(G)$, $[X, Y]$ denotes the set of arcs of G with one end in X and one end in Y. If the head of the arc $a \in [X, Y]$ is in Y, we call a a *forward arc* (of $[X, Y]$); otherwise, we call it a *backward arc*.

The acyclicity of G implies a natural ordering of the vertices into disjoint sets, as follows. We define $S^0(G)$ to denote the set of vertices with in-degree 0 in G (so $S^0(G) = S'(G)$), $S^1(G)$ the set of vertices with in-degree 0 in the graph obtained from G by deleting the vertices of $S^0(G)$ and all arcs with tails in $S^0(G)$, and so on, until the final set $S^t(G)$ contains the remaining vertices with in-degree 0 and out-degree 0 in the remaining graph. Note that these sets are well-defined since G is acyclic, and also note that $S^t(G) \neq S''(G)$, in general. If a vertex $v \in V(G)$ is in the set $S^j(G)$ in the above ordering, we say that v is *at level* j in G. This ordering implies that each arc $a \in A(G)$ can only have $tail(a) \in S^{j_1}(G)$ and $head(a) \in S^{j_2}(G)$ if $j_1 < j_2$.

A graph G is called *weakly connected* if all pairs of distinct vertices u and v of G are connected through a sequence of distinct vertices $u = v_0 v_1 \dots v_k = v$ and arcs $a_1 a_2 \dots a_k$ of G with $\mu(a_i) = (v_{i-1}, v_i)$ or (v_i, v_{i-1}) for $i = 1, 2, \dots, k$. We are mainly interested in weakly connected graphs, or in the weakly connected components of a graph G. If $X \subseteq V(G)$, then the *subgraph of G induced by X*, denoted as $G[X]$, is the graph on vertex set X containing all the arcs of G which have both their ends in X (together with L, μ and λ restricted to this subset of the arcs). If $X \subseteq V(G)$ induces a weakly connected subgraph of G, but there is no set $Y \subseteq V(G)$ such that $G[Y]$ is weakly connected and X is a proper subset of Y, then $G[X]$ is called a *weakly connected component* of G. In the sequel, throughout we omit the words weakly connected, so a component should always be understood

as a weakly connected component. In contrast to the notation in the textbook of Bondy and Murty (2008), we use $\omega(G)$ to denote the number of components of a graph G.

We denote the components of G by G_i, where i ranges from 1 to $\omega(G)$ of G. In that case, we use V_i, A_i and L_i as shorthand notation for $V(G_i)$, $A(G_i)$ and $L(G_i)$, respectively. The mappings μ and λ have natural counterparts restricted to the subsets $A_i \subset A(G)$ that we do not specify explicitly. We use $G = \sum_{i=1}^{\omega(G)} G_i$ to indicate that G is the disjoint union of its components, implicitly defining its components as G_1 up to $G_{\omega(G)}$. In particular, $G = G_1$ if and only if G is weakly connected itself.

A graph G is called *deterministic*[1] if its arcs have the following property. If $\lambda(a) = \lambda(b)$ for two arcs $a \in A$ and $b \in A$ with $head(a) \neq head(b)$, then $tail(a) \neq tail(b)$.

An arc a with label pair $\lambda(a)$ in a graph G is a *synchronising arc* with respect to another graph H, if and only if there exists an arc $b \in A(H)$ with label pair $\lambda(b)$ such that $\lambda(a) = \lambda(b)$.

We assume that two different arcs with the same head and tail have different labels; otherwise, we replace such multiple arcs by one arc with that label, because these arcs represent exactly the same action at the same stage of a process. Hence, we require that the following property holds for all the graphs we consider: any two distinct arcs $a \in A$ and $b \in A$ with $\mu(a) = \mu(b)$ have $\lambda(a) \neq \lambda(b)$.

For each pair $(v_i, v_j) \in V(G) \times V(G)$, we let $A(v_i, v_j) = \{a \in A(G) \mid \mu(a) = (v_i, v_j)\}$, and we let $t_m(v_i, v_j) = \max_{a \in A(v_i v_j)} t(a)$.

A sequence of distinct vertices $v_0 v_1 \dots v_k$ and arcs $a_1 a_2 \dots a_k$ of G is a (directed) path[2] in G if $\mu(a_i) = (v_{i-1}, v_i)$ for $i = 1, 2, \dots, k$. We denote such a path by $P = v_0 a_1 v_1 a_2 \dots a_k v_k$, and we define its weight as $w(P) = \sum_{a_i \in A(P)} t(a_i)$.

A path from a vertex of the source of G to a vertex of the sink of G is called a *full path* (of G).

The *path length* of G_i, denoted by $\ell(G_i)$, is the maximum of $w(P)$ taken over all full paths P of G_i.

The path length of a graph $G = \sum_{i=1}^{\omega(G)} G_i$, denoted by $\ell(G)$, is defined as $\ell(G) = \sum_{i=1}^{\omega(G)} \ell(G_i)$.

In the next section, we introduce a (directed labelled multigraph) analogue of the

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

[2] There is a close relationship between a trace and a directed path; "a trace is a sequence of visible actions in the order they are observed." (Roscoe, 2010, page 29), a trace $b_1 b_2 \dots b_n$ of a process Q is represented by a path $P = v_0 a_1 v_1 \dots v_{n-1} a_n v_n$ in $G, \ell(a_i) = b_i, i = 1, 2, \dots, n$ where the process Q is represented by the graph G.

Cartesian product of two graphs and several other products we derive from it, resulting in what we call the VRSP.

6.1.2 Graph Products

In Chapter 3, we defined the VRSP in five steps, from the disjoint union of graphs, via the Cartesian product, the weak synchronised product, the reduced weak synchronised product to the VRSP. In the remaining part of this thesis, it is more convenient to define the VRSP alternatively, by combining the first three steps of the former definition.

Instead of defining products for general pairs of graphs, for notational reasons we find it convenient to define those products for two components G_i and G_j of a disconnected graph G. We start by introducing the next analogue of the Cartesian product.

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

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

The first step in this modification consists of ignoring (in fact deleting) the synchronising arcs while forming arcs in the product, but additionally combining pairs of synchronising arcs of G_i and G_j into one arc, yielding the *intermediate product* which we denote by $G_i \boxtimes G_j$.

To be more precise, $G_i \boxtimes G_j$ is obtained from $G_i \square G_j$ by first ignoring all except for the so-called *asynchronous* arcs, i.e., by only maintaining all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_j = w_j$ and $\lambda(a) \notin L_j$, as well as all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_i = w_i$ and $\lambda(a) \notin L_i$. Additionally, we add arcs that replace synchronising pairs $a_i \in A_i$ and $a_j \in A_j$ with $\lambda(a_i) = \lambda(a_j)$. If $\mu(a_i) = (v_i, w_i)$ and $\mu(a_j) = (v_j, w_j)$, such a pair is replaced by an arc $a_{i,j}$ with $\mu(a_{i,j}) = ((v_i, v_j), (w_i, w_j))$ and $\lambda(a_{i,j}) = \lambda(a_i)$. We call such arcs of $G_i \boxtimes G_j$ *synchronous* arcs.

The second step in this modification consists of removing (from $G_i \boxtimes G_j$) the

vertices $(v_i, v_j) \in V_{i,j}$ and the arcs a with $tail(a) = (v_i, v_j)$, in the case that (v_i, v_j) has $level > 0$ in $G_i \square G_j$ but $level\, 0$ in $G_i \boxtimes G_j$. This is then repeated in the newly obtained graph, and so on, until there are no more vertices at $level\, 0$ in the current graph that are at $level > 0$ in $G_i \square G_j$. This finds its motivation in the fact that in our applications, the states that are represented by such vertices can never be reached, so are irrelevant.

The resulting graph is called the *vertex-removing synchronised product* (VRSP for short) of G_i and G_j, and denoted as $G_i \boxtimes G_j$. For $k \geqslant 3$, the VRSP $G_1 \boxtimes G_2 \boxtimes \cdots \boxtimes G_k$ is defined recursively as $((G_1 \boxtimes G_2) \boxtimes \cdots) \boxtimes G_k$. The VRSP is commutative, but not associative in general, in contrast to the Cartesian product. However, as we will observe later in this chapter, associativity of the VRSP is guaranteed if we require the graphs on which we apply the VRSP to be pairwise consistent, a notion we are going to define in Section 6.2.

In our analysis and exploitation of synchronising processes, it is crucial that we consider situations that are feasible with respect to time and memory constraints. This requires that we look at different combinations of the set of all processes, allowing several subsets of synchronised processes to run in parallel. Translated to products of the components $G_1, G_2, \ldots, G_{\omega(G)}$, of a graph G, we need the concept of a disjoint union of the VRSP of subsets of the set of all components. To be more precise, we consider a partition $\pi = \{I_1, I_2, \ldots, I_s\}$ of the index set $I = \{1, 2, \ldots, \omega(G)\}$, and let J_i denote a permutation of the elements of I_i for $i = 1, 2, \ldots, s$. We let $G^{\pi}_{\boxtimes} = \sum_{i=1}^{s} \boxtimes_{j \in J_i} G_j$ denote the disjoint union of s graphs, each of which is obtained by taking the VRSP of the components of G indexed by a permutation J_i of I_i, for $i = 1, 2, \ldots, s$. If $J_i = \{j_1, j_2, \ldots, j_k\}$, then $\boxtimes_{j \in J_i} G_j = G_{j_1} \boxtimes G_{j_2} \boxtimes \cdots \boxtimes G_{j_k}$. Note that we do not assume that $j_1 < j_2 < \ldots < j_k$. This reflects that in case $\boxtimes$ is non-associative, $G_{j_1} \boxtimes G_{j_2} \boxtimes G_{j_3}$ is in general not isomorphic to $G_{j_2} \boxtimes G_{j_1} \boxtimes G_{j_3}$.

We denote the set of all possible VRSPs of subsets of the components of the graph G as $\mathfrak{S} = \{ \boxtimes_{j \in J_i} G_j \mid J_i \text{ is any permutation of a nonempty subset of } I\}$, and say that $G = \sum_{i=1}^{\omega(G)} G_i$ generates $\mathfrak{S}$.

Note that we allow subsets (index sets) of cardinality 1; in such cases the VRSP has no meaning, but we just take the component associated to this index in the disjoint union.

We let $\mathcal{S}$ denote the set of all possible disjoint unions of VRSPs of the components of G, hence $\mathcal{S} = \bigcup_{\pi \in \Pi} G^{\pi}_{\boxtimes}$, where Π is the set of all partitions of $I = \{1, 2, \ldots, \omega(G)\}$ into nonempty sets. The cardinality of $\mathcal{S}$ is the number of different outcomes $N_{\omega(G)}$ defined in Chapter 5. Such a combination, i.e., an element of $\mathcal{S}$, is called a *solution* if the requirements for timeliness and memory occupancy are met.

In a process algebraic specification, a deadlock means that one or more processes

are not able to continue execution. This may happen when the process has reached its final state and has stopped, or this may happen if there is a cycle of processes waiting for one another or if a process is waiting for a process that has reached its final state. We consider the former as normal system behaviour, whereas the latter two are a deadlock. The case where a process is waiting for a process that has reached its final state is typical for our PHRCPs. An example of such behaviour is shown in Figure 6.4. Therefore a set of processes contains a deadlock if at least one of the processes cannot reach its final state. As a consequence, we define a deadlock for graphs as follows: the VRSP $G_i \boxtimes G_j$ of two graphs G_i and G_j contains a deadlock if and only if there exists a vertex pair $(v_i, v_j) \in V_i \times V_j$ with an out-degree $d^+((v_i, v_j)) = 0$ in $G_i \boxtimes G_j$ and an out-degree $d^+((v_i, v_j)) > 0$ in $G_i \square G_j$.

Recall that our processes are acyclic, but are started again at every period of the PHRCS. Therefore, whenever two processes P_1 and P_2 are consistent, this means that in their parallel execution both processes will reach their set of final states. For the two components G_1 and G_2 representing these two processes, this means that the sinks of G_1 and G_2 must represent the final states of P_1 and P_2. But for $G_1 \boxtimes G_2$ this only makes sense if the sink of $V(G_1 \boxtimes G_2)$ represents the final states of the process $P_{1,2}$ (where $P_{1,2}$ is strongly bisimilar to $P_1 || P_2$). We will introduce several contraction concepts in graphs to describe and analyse consistency of the associated processes. This is explained and formalised by the concepts of a weak contraction, a strong contraction and a pseudopath in the sequel.

6.1.3 Graph Isomorphism and Graph Contraction

The isomorphism we introduce in this section is an analogue of a known concept for unlabelled graphs, but involves statements on the labels.

Formally, an *isomorphism* from G to H consists of two bijections $\phi : V(G) \to V(H)$ and $\psi : A(G) \to A(H)$ such that for all $a \in A(G)$ $\mu(a) = (u, v)$ if and only if $\mu(\psi(a)) = (\phi(u), \phi(v))$ and $\lambda(a) = \lambda(\psi(a))$. Since we assume that two different arcs with the same head and tail have different labels, however, the bijection ψ is superfluous. The reason is that, if (ϕ, ψ) is an isomorphism, then ψ is completely determined by ϕ and the labels. In fact, if (ϕ, ψ) is an isomorphism and $\mu(a) = (u, v)$ for an arc $a \in A(G)$, then $\psi(a)$ is the unique arc $b \in A(H)$ with $\mu(b) = (\phi(u), \phi(v))$ and label $\lambda(b) = \lambda(a)$. Thus, we may define an isomorphism from G to H as a bijection $\phi : V(G) \to V(H)$ such that there exists an arc $a \in A(G)$ with $\mu(a) = (u, v)$ if and only if there exists an arc $b \in A(H)$ with $\mu(b) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$.

We distinguish two types of contractions. The first type contracts vertices in $G_1 \boxtimes G_2$ related to asynchronous arcs of graphs G_1 and G_2 and is called a *weak contraction*. The second type contracts a set of vertices without taking into account whether the arcs belonging to these vertices are synchronous or asynchronous and is called a *strong contraction*.

Let $a \in A(G)$ with $\mu(a) = (u, v)$. By contracting a we mean replacing u and v

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

To define the notion of weak contraction, let T be the set of asynchronous arcs in $G_1 \boxtimes G_2$ that correspond to arcs in G_1. Then the *weak contraction of* $G_1 \boxtimes G_2$ *with respect to* G_1, denoted by $\rho_{G_1}(G_1 \boxtimes G_2)$, is defined as the graph obtained from $G_1 \boxtimes G_2$ by successively contracting each arc $a \in T$. Likewise, let T be the set of asynchronous arcs in $G_1 \boxtimes G_2$ that correspond to arcs in G_2. Then the *weak contraction of* $G_1 \boxtimes G_2$ *with respect to* G_2, denoted by $\rho_{G_2}(G_1 \boxtimes G_2)$, is defined as the graph obtained from $G_1 \boxtimes G_2$ by successively contracting each arc $a \in T$. We also use G_1^ρ as shorthand for $\rho_{G_2}(G_1 \boxtimes G_2)$ and G_2^ρ as shorthand for $\rho_{G_1}(G_1 \boxtimes G_2)$.

Let H be a subgraph of $G_1 \boxtimes G_2$. Then in $\rho_{G_2}(G_1 \boxtimes G_2)$, H corresponds to a subgraph H' of G_1. We denote this H' by $\rho_{G_2}(H)$, and say that H is mapped to $\rho_{G_2}(H)$ by ρ_{G_2}. We use similar terminology and notation with respect to for $\rho_{G_1}(H)$.

We now turn to the definition of strong contraction. Let X be a nonempty proper subset of $V(G)$, and let $Y = V(G) \backslash X$. Then to obtain the strong contraction of G with respect to X, we first replace X by a new vertex $\tilde{x}$, deleting all arcs with both ends in X, delete all arcs $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in X$ and $v \in Y$ by an arc c with $\mu(c) = (\tilde{x}, v)$ and $\lambda(c) = \lambda(a)$, and replace each arc $b \in A(G)$ with $\mu(b) = (u, v)$ for $u \in Y$, and replace $v \in X$ by an arc d with $\mu(d) = (u, \tilde{x})$ and $\lambda(d) = \lambda(b)$. If after this contraction there are arcs with the same ends and labels, then these arcs are replaced by one arc with the same ends and label. We denote the resulting graph as G/X, and say that G/X is the *strong contraction of* G *with respect to* X.

We use the strong contraction in particular to remove non-determinism, in the following way. Recall that non-determinism occurs in a graph G if there is a set of arcs $B \in A(G)$ with the same tail and label, but different heads. In this case, let us denote such a set of different heads by Z. In G/Z, all the arcs of B (with heads in Z) are replaced by one arc with the same tail and label and a new head. So, this removes the non-determinism from G caused by the arc set B. If there occurs non-determinism in the graph G/Z, we iteratively repeat the above contraction procedure until the resulting graph is deterministic. We denote the resulting graph by G^δ.

Let H be a subgraph of G. Then in G^δ, the graph that corresponds to H is denoted by H^δ. We say that H is mapped to H^δ by δ.

The above two types of contractions play a key role in our notion of consistency of graphs. Before we define this notion, we first introduce one additional concept. This concept relates paths in $G_1 \boxtimes G_2$ to paths in $(G_1^\rho)^\delta$ and $(G_2^\rho)^\delta$, in the following way.

A full path P of $G_1 \boxtimes G_2$ is called a *pseudopath* if $\rho_{G_2}(P)$ is isomorphic to a full path in G_1, and $\rho_{G_1}(P)$ is isomorphic to a full path in G_2. Note that in this case $(\rho_{G_2}(P))^\delta$ is a unique full path in $(G_1^\rho)^\delta$, and $(\rho_{G_1}(P))^\delta$ is a unique full path in $(G_2^\rho)^\delta$, that satisfy this condition. In particular, P is a full path in $(\rho_{G_2}(P))^\delta \boxtimes (\rho_{G_1}(P))^\delta$. We often say there exists a full path in G_1 (G_2) for a pseudopath in $G_1 \boxtimes G_2$ if we mean that these paths exist in the above sense when $(G_1^\rho)^\delta \cong G_1$ (and $(G_2^\rho)^\delta \cong G_2$). Similarly, we often say there exists a pseudopath in $G_1 \boxtimes G_2$ for every full path in G_1 (G_2) if we mean that there exists a pseudopath P in $Q \boxtimes R$ for full paths Q in G_1 and R in G_2.

As an example, Figure 6.1 shows the pseudopath $(v_1, w_1)c(v_2, w_1)a(v_2, w_2)s(v_3, w_3)$ $b(v_4, w_3)d(v_4, w_4)$ in $G_1 \boxtimes G_2$ and the full paths $v_1 c v_2 s v_3 b v_4$ in G_1 and $w_1 a w_2 s$ $w_3 d w_4$ in G_2. In this example the dashed arc in $G_1 \boxtimes G_2$ is a synchronous arc and the result of the synchronising, dashed arcs in both components G_1 and G_2. The dotted arcs in G_1 and the normal (not dotted and not dashed) arcs in G_2 are not-synchronising arcs.

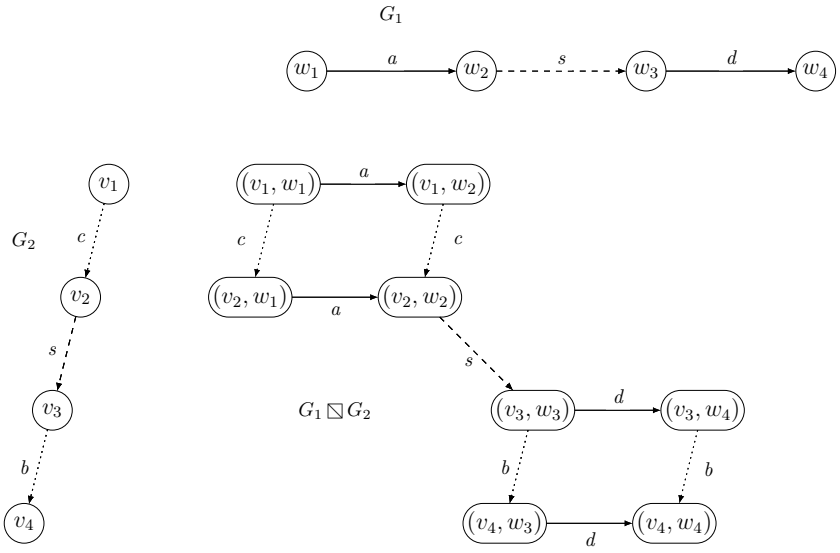


Figure 6.1: Full paths in G_1 and G_2, and pseudopaths in $G_1 \boxtimes G_2$.

6.2 Consistency of Graphs under the VRSP

In this section, we introduce the concept of consistency of graphs, formalised in Definition 6.2.1. With respect to processes this concept is vital; processes that are not consistent contain deadlocks which violate the safety requirements of a PHRCS. The lack of consistency is therefore unacceptable in a PHRCS.

It is natural to look for a notion of consistency of graphs that is equivalent to

consistency of the associated processes. We will show that the following definition meets our requirements. As we will see, this definition relies heavily on the two types of contractions.

Definition 6.2.1. *Components G_1 and G_2 are consistent, denoted as $G_1 \doteqdot G_2$, if and only if the following two requirements hold:*

1. $(G_1^\rho)^\delta \cong G_1$ *and* $(G_2^\rho)^\delta \cong G_2$.
2. $S'(G_1 \boxtimes G_2) = S'(G_1) \times S'(G_2)$ *and* $S''(G_1 \boxtimes G_2) = S''(G_1) \times S''(G_2)$.

We will show what the above definition implies for the existence of pseudopaths, and in fact give an equivalent characterisation of consistency of components in terms of pseudopaths. But first we continue with three examples to show that both requirements in the above definition are essential for consistency of the associated processes.

Two examples where the weak contraction of $G_1 \boxtimes G_2$ violates the first requirement are given in Figure 6.2 and Figure 6.3.

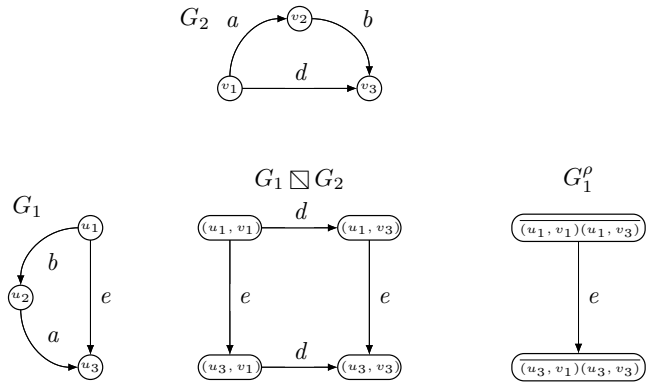


Figure 6.2: Inconsistent components G_1 and G_2 violating requirement 1 ($G_1 \ncong G_1^\rho$).

In Figure 6.2, there does not exist a full path in G_1^ρ isomorphic to the full path $u_1 b u_2 a u_3$ of G_1, therefore the processes P_1 and P_2 represented by $G1$ and G_2, respectively, can never perform either action a or action b. Hence, P_1 and P_2 are inconsistent. When G_1 and G_2 comply with requirement 1, this situation cannot occur. In Figure 6.3, the graph G_1^ρ is a non-deterministic graph, representing a non-deterministic process. Therefore requirement 1 is necessary to eliminate the non-determinism from G_1^ρ.

An example that violates the second requirement is given in Figure 6.4.

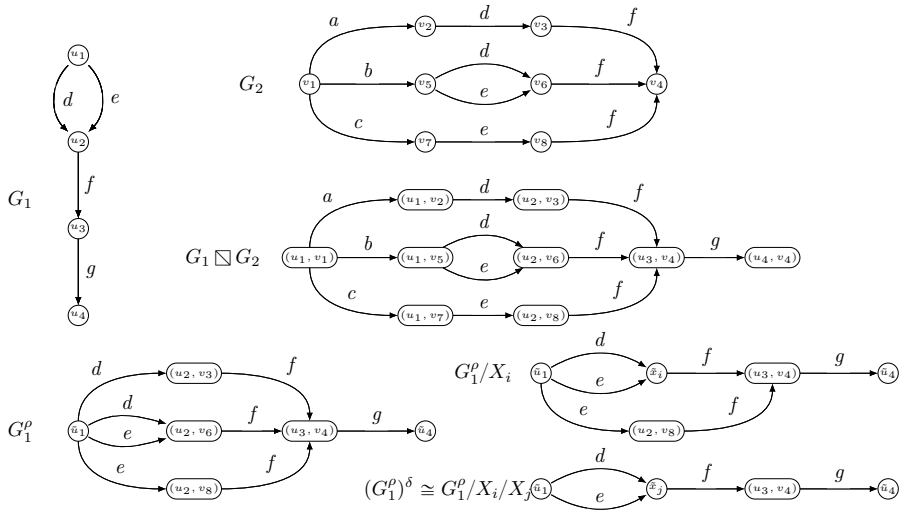


Figure 6.3: Non-deterministic graph G_1^ρ with only a weak contraction of $G_1 \diagdown\!\!\diagdown G_2$, violating requirement 1.

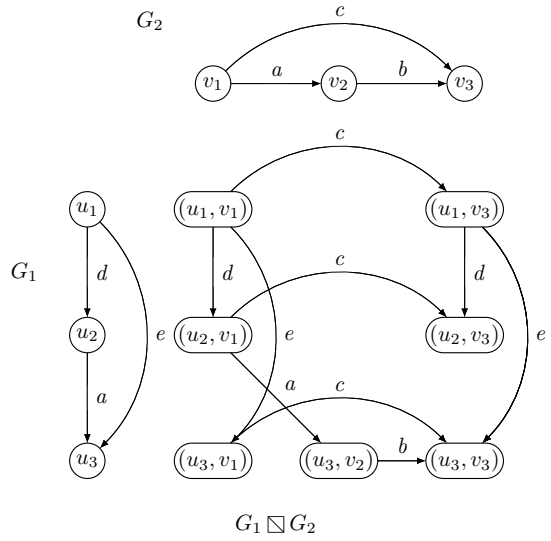


Figure 6.4: Inconsistent components G_1 and G_2 violating requirement 2.

Although the processes P_1 and P_2 represented by, respectively, the graphs G_1 and G_2 in Figure 6.4, seem to be consistent in the sense that every action is part of a

trace of $P_1||P_2$, P_1 suffers from a deadlock if P_2 performs the action with label c represented by the arc c with $\mu(c) = (v_1, v_2)$, $\lambda(c) = c$ in G_2. This situation is prevented if the graphs G_1 and G_2 (and therefore the processes P_1 and P_2) comply with requirement 2.

These examples show clearly that both requirements of consistency are necessary to exclude a deadlock in the processes represented by the components.

Next, we are going to establish an equivalent characterization of consistency of graphs in terms of the existence of pseudopaths. We first make the following observation. Suppose that G_1 and G_2 are consistent, and that P is a full path in G_1 and Q is a full path in G_2. Then, clearly $(\rho_{_Q}(P \boxtimes Q))^\delta \cong P$ and $(\rho_{_P}(P \boxtimes Q))^\delta \cong Q$, and every full path in $P \boxtimes Q$ is thus a pseudopath in $P \boxtimes Q$.

Next, we first state and prove the following consequences of Definition 6.2.1 in terms of the existence of pseudopaths.

Lemma 6.2.2. *Suppose that the components G_1 and G_2 are consistent. Then*

(i) all full paths of $G_1 \boxtimes G_2$ are pseudopaths;

(ii) for any two full paths Q in G_1 and R in G_2 there exists a pseudopath in $G_1 \boxtimes G_2$.

Proof. Let G_1 and G_2 be consistent components of a graph. Then, by Definition 6.2.1, $(G_1^\rho)^\delta \cong G_1$ and $(G_2^\rho)^\delta \cong G_2$, and $S'(G_1 \boxtimes G_2) = S'(G_1) \times S'(G_2)$ and $S''(G_1 \boxtimes G_2) = S''(G_1) \times S''(G_2)$.

To prove *(i)*, let P be an arbitrary full path of $G_1 \boxtimes G_2$. It is sufficient to show that P is a pseudopath of $G_1 \boxtimes G_2$, hence that $\rho_{G_2}(P)$ is isomorphic to a full path of G_1. Then by symmetry, $\rho_{G_1}(P)$ is isomorphic to a full path of G_2. From the definition of ρ_{G_2} and δ, we know that P is mapped to a unique path $(\rho_{G_2}(P))^\delta$. Let Q denote the full path in G_1 isomorphic to $(\rho_{G_2}(P))^\delta$. Similarly, let R denote the full path in G_2 isomorphic to $(\rho_{G_1}(P))^\delta$. Then P is isomorphic to a path in $Q \boxtimes R$. If Q is not a full path in G_1, then clearly P is not a full path in $Q \boxtimes R$. This contradicts the assumptions and completes the proof of *(i)*.

To prove *(ii)*, let Q be a full path of G_1, and R be a full path of G_2. Consider any full path P of $Q \boxtimes R$. It is sufficient to show that P is a pseudopath of $Q \boxtimes R$. This follows immediately from *(i)*. □

We now show that the converse of Lemma 6.2.2 also holds. Hence, we obtain a characterisation of consistency in terms of pseudopaths.

Lemma 6.2.3. *Let G_1 and G_2 be two components of a graph G. If all full paths in $G_1 \boxtimes G_2$ are pseudopaths, and for any two full paths Q in G_1 and R in G_2 there exists a pseudopath in $G_1 \boxtimes G_2$, then G_1 and G_2 are consistent.*

Proof. Let G_1 and G_2 be components of a graph G satisfying *(i)* and *(ii)* of Lemma 6.2.2. We are going to show that G_1 and G_2 are consistent by proving that G_1 and G_2 satisfy the two requirements of Definition 6.2.1.

We first claim that $S'(G_1 \diagdown\!\!\diagdown G_2) = S'(G_1) \times S'(G_2)$ and $S''(G_1 \diagdown\!\!\diagdown G_2) = S''(G_1) \times S''(G_2)$. Let $(v, w) \in S'(G_1) \times S'(G_2)$ and $(p, q) \in S''(G_1) \times S''(G_2)$. Then there exists a full path Q in G_1 from v to p, and there exists a full path R in G_2 from w to q. By *(ii)* of Lemma 6.2.2 there exists a pseudopath P in $G_1 \diagdown\!\!\diagdown G_2$ from (v, w) to (p, q). Hence $(v, w) \in S'(G_1 \diagdown\!\!\diagdown G_2)$ and $(p, q) \in S''(G_1 \diagdown\!\!\diagdown G_2)$. Thus $S'(G_1) \times S'(G_2) \subseteq S'(G_1 \diagdown\!\!\diagdown G_2)$, and $S''(G_1) \times S''(G_2) \subseteq S''(G_1 \diagdown\!\!\diagdown G_2)$. By *(i)* of Lemma 6.2.2, clearly $S'(G_1 \diagdown\!\!\diagdown G_2) \subseteq S'(G_1) \times S'(G_2)$ and $S''(G_1 \diagdown\!\!\diagdown G_2) \subseteq S''(G_1) \times S''(G_2)$. This completes the proof that G_1 and G_2 satisfy requirement 2 of Definition 6.2.1.

We next prove that G_1 and G_2 satisfy requirement 1 of Definition 6.2.1. We already observed that if H is a subgraph of $G_1 \diagdown\!\!\diagdown G_2$, then in $G_1^\rho = \rho_{G_2}(G_1 \diagdown\!\!\diagdown G_2)$, $\rho_{G_2}(H)^\delta$ is isomorphic to a unique subgraph of G_1, and $(G_1^\rho)^\delta$ is isomorphic to a unique subgraph of G_1. To show that $(G_1^\rho)^\delta \cong G_1$, we use the obvious observation that an arbitrary arc a of G_1 is contained in a full path Q of G_1. By *(ii)* of Lemma 6.2.2, for this path Q, there exists a pseudopath P in $G_1 \diagdown\!\!\diagdown G_2$ such that $\rho_{G_2}(P)$ is isomorphic to Q. As we have argued before, P is mapped to a unique path $(\rho_{G_2}(P))^\delta$ in $(G_1^\rho)^\delta$. This implies that $(G_1^\rho)^\delta$ contains a unique arc that is in one-to-one correspondence with the arc a of G_1. Together with the fact that $(G_1^\rho)^\delta$ is isomorphic to a unique subgraph of G_1, this shows that $(G_1^\rho)^\delta \cong G_1$. Similarly, $(G_2^\rho)^\delta \cong G_2$, completing the proof of Lemma 6.2.3. □

In the next section, we show that the consistency of graphs that we defined in the current section corresponds to the consistency of the associated processes.

6.3 The Consistency of Processes Compared with the Consistency of Graphs

In this section, we will show that the consistency of graphs is equivalent to the consistency of the associated processes. Recall that there is a one-to-one correspondence between the processes and graphs that we consider, and also between traces and paths. But first, for convenience, we repeat the relevant definitions of consistency of processes given in Chapter 2.

Let $\mathcal{P}$ be a set of states and let Act be a set of actions. Let τ represent any asynchronous action of either the process P or the process Q.

For a set of states $\mathcal{P}$ with $\alpha \in Act$, $X, X', Y, Y' \in \mathcal{P}$, we write $X \stackrel{\alpha}{\Rightarrow} Y$ if and only if

- if $\alpha \neq \tau$, we have $X \stackrel{\tau^*}{\rightarrow} X' \stackrel{\alpha}{\rightarrow} Y' \stackrel{\tau^*}{\rightarrow} Y$
- if $\alpha = \tau$, we have $X \stackrel{\tau^*}{\rightarrow} Y$, where τ^* stands for a (possibly empty) sequence of τ-labelled transitions.

Then a weak bisimulation is defined as follows:

A binary relation $S \subseteq \mathcal{P} \times \mathcal{P}$ is a weak bisimulation if $(P, Q) \in S$ implies, for all $\alpha \in Act$,

(i) Whenever $P \xrightarrow{\alpha} P'$ then, for some $Q', Q \stackrel{\alpha}{\Rightarrow} Q'$ and $(P', Q') \in S$

(ii) Whenever $Q \xrightarrow{\alpha} Q'$ then, for some $P', P \stackrel{\alpha}{\Rightarrow} P'$ and $(P', Q') \in S$

When there is a weak or strong bisimulation between two processes, we call these two processes weakly or strongly bisimilar. The processes P and Q are consistent if and only if they are weakly bisimilar, denoted as $P \approx Q$.

We continue with the relevant notions for traces of processes with respect to paths of graphs. We call a trace T of a process P equivalent to a path Q of a graph G if there is a one-to-one correspondence between the actions of the trace T and the labels of the arcs of the path Q, preserving the order of these actions and these labels, i.e., if T is a trace $(l_1, t_1)(l_2, t_2) \dots (l_n, t_n)$ then Q is a path $u_0 a_1 u_1 a_2 u_2 \dots u_{n-1} a_n u_n$, with $\lambda(a_i) = (l_i, t_i)$, $i = 1, \dots, n$, and if Q is a path $u_0 a_1 u_1 a_2 u_2 \dots u_{n-1} a_n u_n$, with $\lambda(a_i) = (l_i, t_i)$, $i = 1, \dots, n$, then T is a trace $(l_1, t_1)(l_2, t_2) \dots (l_n, t_n)$.

We define a trace from a begin state to a final state of a process P as a *full trace*. With the definition of a full trace of a process we have the equivalence of a full path in a graph.

A *pseudotrace* T in $P_1||P_2$ is a full trace in $P_1||P_2$ for which there exist a full trace T_1 in P_1, a full trace T_2 in P_2 such that T is a full trace in $T_1||T_2$. Note that, although we do not construct a process $P \sim P_1||P_2$, the notion of a pseudotrace is equivalent to the notion of a pseudopath.

Firstly, similar to the weak contraction of graphs, we map a process P which is strongly bisimilar to $P_1||P_2$ onto the processes P_1' and P_2'. We achieve this by removing all silent actions of P_2 from P which gives P_1', and by removing all silent actions of P_1 from P which gives P_2'. Each action α has a from-state and a to-state, where the action α is a transition from the from-state to the to-state. The removal of a silent action is in fact the contraction of the from-state and the to-state into one new state, thereby removing the silent action from the process. This contraction is similar to the weak contraction of a graph and is denoted by the function ζ which maps a process onto a process, $\zeta_{P_2}(P) = P_1'$ and $\zeta_{P_1}(P) = P_2'$. Note that if there is both a silent action and a non-silent action from some from-state to a to-state, the removal of the silent action will lead to a loop of the non-silent action. But this cannot happen for consistent processes.

Secondly, in the same manner we can map a process that is not deterministic onto a process that is deterministic by contracting the to-states of actions with the same from-state and identical labels. This contraction is then similar to the strong contraction of a graph and is denoted by the function ξ, defined by $\xi(P_1') = P_1''$ and $\xi(P_2') = P_2''$.

Finally, we define the function ϑ that maps a process $P \sim P_1||P_2$ onto its constituent processes P_1'' and P_2'' as $\vartheta_{P_2}(P) = \xi(\zeta_{P_2}(P)) = P_1''$ and $\vartheta_{P_1}(P) = \xi(\zeta_{P_1}(P)) = P_2''$.

Note that for non-consistent processes P_1 and P_2 the processes P_1'' or P_2'' are not strongly bisimilar to P_1 or P_2, respectively, and that for consistent processes P_1 and P_2 the processes P_1'' or P_2'' are strongly bisimilar to P_1 or P_2, respectively.

Because there is a one-to-one correspondence between a process P and the graph G representing this process P, there exists a bijective mapping T that transforms a process P into a process G. Then we have for consistent graphs G_1 and G_2 representing the processes P_1 and P_2, respectively, the following series of mappings:

$$\begin{array}{ccc} P_1||P_2 & \stackrel{T}{\rightarrow} & G_1 + G_2 \\ \uparrow \vartheta & & \downarrow \boxtimes \\ P & \stackrel{T^{-1}}{\leftarrow} & G_1 \boxtimes G_2 \end{array}$$

Because of the one-to-one correspondence of full traces and full paths, pseudotraces and pseudopaths and the equivalence of the function ϑ for processes and the weak and strong contraction of graphs, we have that, without repeating the proofs for processes similar to the consistency related proofs for graphs, consistency for graphs is equivalent to consistency for processes.

6.4 Associativity of the VRSP

We already noticed before that the VRSP is not associative in general. However, if the components are strongly pairwise consistent, in the sense we are going to define next, the picture changes. In Theorem 6.4.1, we show that the VRSP is associative over a set of strongly pairwise consistent components.

Let G be a graph generating $\mathfrak{S}$. Then we say that the components of G are *strongly pairwise consistent* if all disjoint components of $\mathfrak{S}$ are pairwise consistent. We say that the VRSP is *associative* over a set of components S if $(G_1 \boxtimes G_2) \boxtimes G_3$ is isomorphic to $G_1 \boxtimes (G_2 \boxtimes G_3)$ for all disjoint components G_1, G_2, and G_3 of S.

Theorem 6.4.1. *The VRSP is associative over any set of strongly pairwise consistent components.*

Proof. Let G_1, G_2, and G_3 be any triple of disjoint components from a set of strongly pairwise consistent components. By Requirement 2 of Definition 6.2.1, $S'(G_1 \boxtimes G_2) = S'(G_1) \times S'(G_2)$ and $S''(G_1 \boxtimes G_2) = S''(G_1) \times S''(G_2)$. Together with the fact that the Cartesian product is associative over sets, this immediately shows that $S'((G_1 \boxtimes G_2) \boxtimes G_3) = S'(G_1 \boxtimes (G_2 \boxtimes G_3))$ and $S''((G_1 \boxtimes G_2) \boxtimes G_3) = S''(G_1 \boxtimes (G_2 \boxtimes G_3))$. Clearly, both $(G_1 \boxtimes G_2) \boxtimes G_3$ and $G_1 \boxtimes (G_2 \boxtimes G_3)$ have vertex sets corresponding to subsets of $V(G_1) \times V(G_2) \times V(G_3)$. It is sufficient to show that every arc in $(G_1 \boxtimes G_2) \boxtimes G_3$ is in one-to-one correspondence with an arc in $G_1 \boxtimes (G_2 \boxtimes G_3)$.

Let a be an arbitrary arc in $(G_1 \boxtimes G_2) \boxtimes G_3$ with $\mu(a) = (((u_1, u_2), u_3), ((v_1, v_2), v_3))$. Clearly, a lies on a full path P in $(G_1 \boxtimes G_2) \boxtimes G_3$. By Lemma 6.2.3, P is a pseudopath in $(G_1 \boxtimes G_2) \boxtimes G_3$, and there exist full paths R in $G_1 \boxtimes G_2$ and Q_3 in G_3, such

that P is a path in $R \boxtimes Q_3$. Similarly, R is a pseudopath in $G_1 \boxtimes G_2$, so there exist full paths Q_1 and Q_2 in G_1 and G_2, respectively, such that P is a path in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$.

We distinguish two cases, and for each case two subcases, depending on whether a is the result of synchronising arcs in G_1, G_2 and G_3 or not.

In the proofs for the subcases, we often use p to denote an arc in Q_1 with $\mu(p) = (u_1, v_1)$, q to denote an arc in Q_2 with $\mu(q) = (u_2, v_2)$, and r to denote an arc in Q_3 with $\mu(r) = (u_3, v_3)$.

Case 1 a is an asynchronous arc of $(Q_1 \boxtimes Q_2) \boxtimes Q_3$. There are two subcases:

Case 1.1 a is the result of an asynchronous arc x_{pq} in $Q_1 \boxtimes Q_2$ and an arc r in Q_3.

In this subcase, it is obvious that a is an arc in the Cartesian product $(Q_1 \square Q_2) \square Q_3$. Hence, because the Cartesian product is associative, there is nothing to prove.

Case 1.2 a is the result of a synchronous arc x_{pq} in $Q_1 \boxtimes Q_2$ and an arc r in Q_3.

In this subcase, $\mu(x_{pq}) = ((u_1, u_2), (v_1, v_2))$ for two arcs p and q with $\lambda(x_{pq}) = \lambda(p) = \lambda(q)$. Furthermore, $\lambda(r) \neq \lambda(x_{pq})$.

For the arcs x_{pq} and r we have a quadrangle of arcs x_{pqr_1} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_1}) = (((u_1, u_2), u_3), ((u_1, u_2), v_3)$ and x_{pqr_2} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_2}) = (((v_1, v_2), u_3), ((v_1, v_2), v_3)$ and $\lambda(r) = \lambda(x_{pqr_1}) = \lambda(x_{pqr_2})$, and we have arcs x_{pqr_3} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_3}) = (((u_1, u_2), u_3), ((v_1, v_2), u_3)$ and x_{pqr_4} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_4}) = (((u_1, u_2), v_3), ((v_1, v_2), v_3)$ and $\lambda(x_{pq}) = \lambda(x_{pqr_3}) = \lambda(x_{pqr_4})$.

For the arcs q and r we have a quadrangle of arcs x_{qr_1} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr_1}) = ((u_2, u_3), (u_2, v_3))$ and $\lambda(r) = \lambda(x_{qr_1})$, x_{qr_2} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr_2}) = ((v_2, u_3), (v_2, v_3))$ and $\lambda(r) = \lambda(x_{qr_2})$, x_{qr_3} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr_3}) = ((u_2, u_3), (v_2, u_3))$ and $\lambda(q) = \lambda(x_{qr_3})$, and x_{qr_4} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr_4}) = ((u_2, v_3), (v_2, v_3))$ and $\lambda(q) = \lambda(x_{qr_4})$.

For the arcs x_{qr_1} and p we have the arc x_{qrp_1} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_1}) = (((u_2, u_3), u_1), ((u_2, v_3), u_1))$ and $\lambda(r) = \lambda(x_{qrp_1})$.

For the arcs x_{qr_2} and p we have the arc x_{qrp_2} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_2}) = (((v_2, u_3), v_1), ((v_2, v_3), v_1))$ and $\lambda(r) = \lambda(x_{qrp_2})$.

For the arcs x_{qr_3} and p we have the arc x_{qrp_3} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_3}) = (((u_2, u_3), u_1), ((v_2, u_3), v_1))$ and $\lambda(p) = \lambda(x_{qrp_3})$.

For the arcs x_{qr_4} and p we have the arc x_{qrp_4} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_4}) = (((u_2, v_3), u_1), ((v_2, v_3), v_1))$ and $\lambda(p) = \lambda(x_{qrp_4})$.

Because $(Q_2 \boxtimes Q_3) \boxtimes Q_1 \cong Q_1 \boxtimes (Q_2 \boxtimes Q_3)$, and there is a bijection from $V((Q_1 \boxtimes Q_2) \boxtimes Q_3)$ to $V(Q_1 \boxtimes (Q_2 \boxtimes Q_3))$, we have a one-to-one correspondence between the arcs x_{pqr_i} and x_{qrp_i} (for $i = 1, \ldots, 4$).

Since there clearly is a one-to-one correspondence between x_{pqr_i} and x_{qrp_i}, $i = 1, \ldots, 4$, this is in particular the case for a.

Case 2 a is a synchronous arc of $(Q_1 \boxtimes Q_2) \boxtimes Q_3$. There are two subcases:

Case 2.1 a is the result of an asynchronous arc x_{pq} in $Q_1 \boxtimes Q_2$ and an arc r in Q_3.

In this subcase, we have the quadrangle of arcs x_{pq_1} with $\mu(x_{pq_1}) = ((u_1, u_2), (v_1, u_2))$ and x_{pq_2} with $\mu(x_{pq_2}) = ((u_1, v_2), (v_1, v_2))$ with $\lambda(x_{pq_1}) = \lambda(x_{pq_2}) = \lambda(p)$, and x_{pq_3} with $\mu(x_{pq_3}) = ((u_1, u_2), (u_1, v_2))$ and x_{pq_4} with $\mu(x_{pq_4}) = ((v_1, u_2), (v_1, v_2))$ with $\lambda(x_{pq_3}) = \lambda(x_{pq_4}) = \lambda(q)$. Furthermore, either $\lambda(p) = \lambda(r)$ or $\lambda(q) = \lambda(r)$.

Firstly, we are going to prove this subcase for $\lambda(p) = \lambda(r)$.

For the arc x_{pq_1} and r we have arcs x_{pqr_1} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_1}) = (((u_1, u_2), u_3), ((v_1, u_2), v_3)$ and x_{pqr_2} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_2}) = (((u_1, v_2), u_3), ((v_1, v_2), v_3)$ and $\lambda(r) = \lambda(x_{pqr_1}) = \lambda(x_{pqr_2})$, and we have arcs x_{pqr_3} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_3}) = (((u_1, u_2), u_3), ((u_1, v_2), u_3)$ and x_{pqr_4} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_4}) = (((v_1, u_2), v_3), ((v_1, v_2), v_3)$ and $\lambda(q) = \lambda(x_{pqr_3}) = \lambda(x_{pqr_4})$.

For the arcs q and r we have a quadrangle of arcs x_{qr_1} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr_1}) = ((u_2, u_3), (u_2, v_3))$ and $\lambda(r) = \lambda(x_{qr_1})$, x_{qr_2} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr_2}) = ((v_2, u_3), (v_2, v_3))$ and $\lambda(r) = \lambda(x_{qr_2})$, x_{qr_3} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr_3}) = ((u_2, u_3), (v_2, u_3))$ and $\lambda(q) = \lambda(x_{qr_3})$, and x_{qr_4} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr_4}) = ((u_2, v_3), (v_2, v_3))$ and $\lambda(q) = \lambda(x_{qr_4})$.

For the arcs x_{qr_1} and p we have the arc x_{qrp_1} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_1}) = (((u_2, u_3), u_1), ((u_2, v_3), v_1))$ and $\lambda(r) = \lambda(x_{qrp_1})$.

For the arcs x_{qr_2} and p we have the arc x_{qrp_2} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_2}) = (((v_2, u_3), u_1), ((v_2, v_3), v_1))$ and $\lambda(r) = \lambda(x_{qrp_2})$.

For the arcs x_{qr_3} and p we have the arc x_{qrp_3} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_3}) = (((u_2, u_3), u_1), ((v_2, u_3), v_1))$ and $\lambda(q) = \lambda(x_{qrp_3})$.

For the arcs x_{qr_4} and p we have the arc x_{qrp_4} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_4}) = (((u_2, v_3), v_1), ((v_2, v_3), v_1))$ and $\lambda(q) = \lambda(x_{qrp_4})$.

Since there clearly is a one-to-one correspondence between x_{pqr_i} and x_{qrp_i}, $i = 1, \ldots, 4$, this is in particular the case for a.

Secondly, we are going to prove this subcase for $\lambda(q) = \lambda(r)$.

For the arc x_{pq_1} and r we have arcs x_{pqr_1} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_1}) = (((u_1, u_2), u_3), ((u_1, v_2), v_3)$ and x_{pqr_2} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_2}) = (((v_1, u_2), u_3), ((v_1, v_2), v_3)$ and $\lambda(r) = \lambda(x_{pqr_1}) = \lambda(x_{pqr_2})$, and we have arcs x_{pqr_3} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_3}) = (((u_1, u_2), u_3), ((v_1, u_2), u_3)$ and x_{pqr_4} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr_4}) = (((u_1, v_2), v_3), ((v_1, v_2), v_3)$ and $\lambda(p) = \lambda(x_{pqr_3}) = \lambda(x_{pqr_4})$.

For the arcs q and r we have the arc x_{qr} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr}) = ((u_2, u_3), (v_2, v_3))$ and $\lambda(r) = \lambda(x_{qr})$.

For the arc x_{qr} and p we have the quadrangle of arcs x_{qrp_1} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_1}) = (((u_2, u_3), u_1), ((v_2, v_3), u_1))$, x_{qrp_2} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_2}) = (((u_2, u_3), v_1), ((v_2, v_3), v_1))$ and $\lambda(r) = \lambda(x_{qrp_1}) = \lambda(x_{qrp_2})$, and $\mu(x_{qrp_3}) = (((u_2, u_3), u_1), ((u_2, u_3), v_1))$, x_{qrp_3} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_4}) = (((v_2, v_3), u_1), ((v_2, v_3), v_1))$ and $\lambda(p) = \lambda(x_{qrp_3} = \lambda(x_{qrp_4})$.

Because $(Q_2 \boxtimes Q_3) \boxtimes Q_1 \cong Q_1 \boxtimes (Q_2 \boxtimes Q_3)$, and there is a bijection from $V((Q_1 \boxtimes Q_2) \boxtimes Q_3)$ to $V(Q_1 \boxtimes (Q_2 \boxtimes Q_3))$, we have a one-to-one correspondence between the arcs x_{pqr_i} and x_{qrp_i} (for $i = 1, \cdots, 4$).

Since there clearly is a one-to-one correspondence between x_{pqr_i} and x_{qrp_i}, $i = 1, \ldots, 4$, this is in particular the case for a.

Case 2.2 a is the result of a synchronous arc x_{pq} in $Q_1 \boxtimes Q_2$ and an arc r in Q_3.

In this subcase, $\mu(x_{pq}) = ((u_1, u_2), (v_1, v_2))$ for two arcs p and q with $\lambda(x_{pq}) = \lambda(p) = \lambda(q) = \lambda(r)$.

For the arcs x_{pq} and r we have an arc x_{pqr} in $(Q_1 \boxtimes Q_2) \boxtimes Q_3$ with $\mu(x_{pqr}) = (((u_1, u_2), u_3), ((v_1, v_2), v_3)$ and $\lambda(r) = \lambda(x_{pqr})$.

For the arcs q and r we have an arc x_{qr} in $Q_2 \boxtimes Q_3$ with $\mu(x_{qr}) = ((u_2, u_3), (v_2, v_3))$ and $\lambda(r) = \lambda(x_{qr})$.

For the arcs x_{qr} and p we have the arc x_{qrp} in $(Q_2 \boxtimes Q_3) \boxtimes Q_1$ with $\mu(x_{qrp_1}) = (((u_2, u_3), u_1), ((v_2, v_3), v_1))$ and $\lambda(r) = \lambda(x_{qrp})$.

Because $(Q_2 \boxtimes Q_3) \boxtimes Q_1 \cong Q_1 \boxtimes (Q_2 \boxtimes Q_3)$, and there is a bijection from $V((Q_1 \boxtimes Q_2) \boxtimes Q_3)$ to $V(Q_1 \boxtimes (Q_2 \boxtimes Q_3))$, we have a one-to-one correspondence between the arcs x_{pqr} and x_{qrp}.

Since there clearly is a one-to-one correspondence between x_{pqr} and x_{qrp}, this is in particular the case for a.

In all possible cases, we have shown that a is in one-to-one correspondence with an arc a' in $G_1 \boxtimes (G_2 \boxtimes G_3)$ with $\mu(a') = ((u_1, (u_2, u_3)), (v_1, (v_2, v_3)))$ and $\lambda(a) = \lambda(a')$. This completes the proof of Theorem 6.4.1. □

6.5 Discussion and Conclusions

Consistency is essential for a set of processes P that are represented by a graph G. Firstly, consistency guarantees that there are no deadlocks in the set of processes represented by the graph G, which is of vital importance to PHRCS. Secondly, consistency guarantees associativity and therefore (together with commutativity) we can multiply the elements of a set of graphs by the VRSP in any order and achieve the same result. This is especially important for the heuristics we developed, because they may calculate the VRSP of the same (sub)set of graphs in different order. Thirdly, the number of possible outcomes when the VRSP is associative follows the Bell number, which is a magnitude smaller than the Bessel number (in case of non-associativity). Fourthly, our view on associativity has led to the notion of consistency of a set of components of a graph G. Via contractions we have shown that sets of pairwise-consistent components are deadlock free, in a

sense that differs slightly from process algebra. Finally, we are now sure that the heuristics we have developed will calculate isomorphic graphs when multiplied by the VRSP. Furthermore, for a limited set of graphs we can calculate all different outcomes and we are certain that if a solution exists, we will find it.

But what if we have a set of processes for which there does not exist a solution using the VRSP? One way out, for which we introduce and develop the tools in the next chapter, is to use the VRSP in order to enable new combinations of subprocesses of the original set of processes, without changing the functionality and behaviour of the total set of new (sub)processes.

7

The Decomposition by the VRSP

This chapter is based on our paper Boode and Broersma (submitted). Our research is based on partitioning the components of a graph G in such a manner that the multiplication by the VRSP of the components in each partition results in a timely execution of the processes represented by these partitions. It is possible that there does not exist a partition for such a graph G that meets the requirements with respect to the deadline or memory occupancy. But it might be beneficial if we are able to decompose the components for the following reason: when one or more components of the graph G are decomposed, new partitions are possible and it could well be that in the set of new partitions a solution exists that did not exist in the original set of partitions of G. Therefore, we investigate whether decompositions are possible and we introduce two decomposition theorems.

The decomposition of graphs has been extensively researched on unlabelled undirected graphs in, for example, Hammack et al. (2011). For instance, Sabidussi (1960) has shown that every finite connected undirected unlabelled graph can be decomposed uniquely into its prime factors with respect to the Cartesian product. In our case, the graphs are finite, deterministic, labelled, acyclic, directed multigraphs, of which we show that under certain conditions they can be decomposed into smaller graphs. In fact, we show that any undirected unlabelled graph of sufficient size can be labelled in such a manner that such a graph can be decomposed into smaller graphs.

7.1 The First Decomposition Result

We start this section by presenting and proving our first decomposition theorem, of which an illustrative small example is given in Figure 7.1. We assume that the graphs we want to decompose are connected; if not, we can apply our decomposition result to the components separately. Let us start by explaining the example of Figure 7.1 first.

At the top of Figure 7.1 we depicted a small graph G, together with a partition of its vertex set into two nonempty sets X and Y, such that $[X, Y]$ contains forward arcs only. Note that in the figure we indicated labels (label pairs) $\lambda(a)$, etc. just by a, etc. The rest of the figure shows the graph G/Y on the left, the graph G/X

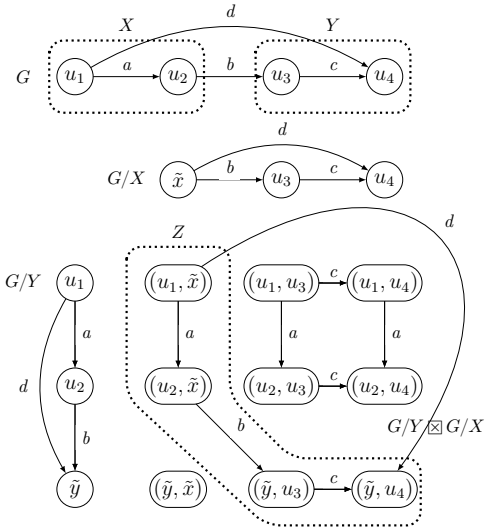


Figure 7.1: Decomposition of $G \cong G/Y \boxtimes G/X$. The set Z from the proof of Theorem 7.1.1 and the graph isomorphic to G induced by Z in $G/Y \boxtimes G/X$ are indicated within the dotted region (except for the arc with label d).

at the top, and the graph $G/Y \boxtimes G/X$; this graph is the result of maintaining the asynchronous arcs and replacing the synchronising arcs by synchronous arcs in the Cartesian product $G/Y \square G/X$. The set of vertices that remain after the second step in the modification are indicated by Z (the vertices in the dotted region). In this example, it is clear that Z induces a graph isomorphic to G. So, in this example $G \cong G/Y \boxtimes G/X$. Our first decomposition theorem states sufficient conditions for this conclusion to hold in general. We will show that none of these conditions can be omitted without violating the conclusion.

Theorem 7.1.1. *Let G be a graph, let X be a nonempty proper subset of $V(G)$, and let $Y = V(G) \backslash X$. Suppose that all the arcs of $[X, Y]$ have distinct labels and that the arcs of G/X and G/Y corresponding to the arcs of $[X, Y]$ are the only synchronising arcs of G/X and G/Y. If $S'(G) \subseteq X$ and $[X, Y]$ has no backward arcs, then $G \cong G/Y \boxtimes G/X$.*

Proof. It clearly suffices to define a mapping $\phi : V(G) \to V(G/Y \boxtimes G/X)$ and to prove that ϕ is an isomorphism from G to $G/Y \boxtimes G/X$.

Let $\tilde{x}$ and $\tilde{y}$ be the new vertices replacing the sets X and Y when defining G/X and G/Y, respectively. Consider the mapping $\phi : V(G) \to V(G/Y \boxtimes G/X)$ defined by

$\phi(v) = (v, \tilde{x})$ for all $v \in X$ and $\phi(w) = (\tilde{y}, w)$ for all $w \in Y$.

Then ϕ is obviously a bijection if $V(G/Y \boxtimes G/X) = Z$, where Z is defined as

$Z = \{(v, \tilde{x}) \mid v \in X\} \cup \{(\tilde{y}, w) \mid w \in Y\}$. We are going to show this later by arguing that all the other vertices of $G/Y \square G/X$ will disappear from $G/Y \boxtimes G/X$. But first, we are going to prove the following claim.

Claim 7.1.2. *The subgraph of $G/Y \boxtimes G/X$ induced by Z is isomorphic to G.*

Proof. Obviously, ϕ is a bijection from $V(G)$ to Z. It remains to show that this bijection preserves the arcs and their labels. By the definition of the Cartesian product, for each arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in X$ and $v \in X$, there exists an arc b in $G/Y \boxtimes G/X$ with $\mu(b) = ((u, \tilde{x}), (v, \tilde{x})) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$. Likewise, for each arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in Y$ and $v \in Y$, there exists an arc b in $G/Y \boxtimes G/X$ with $\mu(b) = ((\tilde{y}, u), (\tilde{y}, v)) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$. Next, consider an arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in X$ and $v \in Y$. For such an arc, in $G/Y \square G/X$ there exist four arcs with label $\lambda(a)$, namely the arcs with $\mu = ((u, \tilde{x}), (\tilde{y}, \tilde{x}))$, $\mu = ((\tilde{y}, \tilde{x}), (\tilde{y}, v))$, $\mu = ((u, \tilde{x}), (u, v))$, and $\mu = ((u, v), (\tilde{y}, v))$. In $G/Y \boxtimes G/X$, these four arcs are replaced by one arc b with $\mu(b) = ((u, \tilde{x}), (\tilde{y}, v)) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$. Since there are no backward arcs in $[X, Y]$, the above arcs are the only arcs in $G/Y \boxtimes G/X$ induced by the vertices of Z. This completes the proof of Claim 7.1.2. □

We continue with the proof of Theorem 7.1.1. It remains to show that all other vertices of $G/Y \boxtimes G/X$, except for the vertices of Z, disappear from $G/Y \boxtimes G/X$. This is clear for the vertex $(\tilde{y}, \tilde{x})$: all the arcs of $G/Y \square G/X$ corresponding to the arcs of $[X, Y]$ are synchronising arcs of G/Y and G/X, so they disappear from $G/Y \boxtimes G/X$. Hence, $(\tilde{y}, \tilde{x})$ has in-degree 0 (and out-degree 0) in $G/Y \boxtimes G/X$, while it has $level > 0$ in $G/Y \square G/X$. For the other vertices, the argument is as follows.

The vertex set of $G/Y \square G/X$ consists of $Z \cup \{(\tilde{y}, \tilde{x})\}$ and the vertex set $X \times Y$. We will argue that all vertices of $X \times Y$ will eventually disappear from $G/Y \boxtimes G/X$.

First of all, we claim that all $(u, v) \in X \times Y$ have $level > 0$ in $G/Y \square G/X$. This is obvious if u has $level > 0$ in $G[X]$ or v has $level > 0$ in $G[Y]$. Now let $(u, v) \in X \times Y$ such that u has $level\, 0$ in $G[X]$ and v has $level\, 0$ in $G[Y]$. Then the claim follows from the fact that v has at least one in-arc from a vertex in X, since $S'(G) \subseteq X$. In fact, since v has only in-arcs from vertices in X and u has no in-arcs at all, (u, v) has $level\, 0$ in $G/Y \boxtimes G/X$. Hence, all vertices $(u, v) \in X \times Y$ such that u has $level\, 0$ in $G[X]$ and v has $level\, 0$ in $G[Y]$ disappear from $G/Y \boxtimes G/X$, together with all the arcs with tail (u, v) for all such vertices $(u, v) \in X \times Y$. If after this first step there are still vertices of $X \times Y$ left in $G/Y \boxtimes G/X$, we can repeat the above arguments step by step for such remaining vertices $(u, v) \in X \times Y$ for which (u, v) has the lowest level in what has remained from $G/Y \boxtimes G/X$. Since $G/Y \boxtimes G/X$ is acyclic, it is clear that all vertices of $X \times Y$ disappear one by one from $G/Y \boxtimes G/X$. This completes the proof of Theorem 7.1.1. □

We are next going to provide some examples to show that none of the essential conditions in Theorem 7.1.1 can be omitted without violating the conclusion. First

of all, it is clear that we need a proper partition of $V(G)$ into nonempty sets; otherwise, the contractions cannot be carried out and the whole discussion is meaningless. In fact, the result is only meaningful if both of the partite sets have at least two vertices. For the other conditions, we show by small examples that they are essential for the validity of the conclusion.

One of the requirements is that the arcs of $[X, Y]$ have distinct labels. The example in Figure 7.2 clearly shows that we cannot omit this requirement. Note that all the other conditions of Theorem 7.1.1 are met by this example graph.

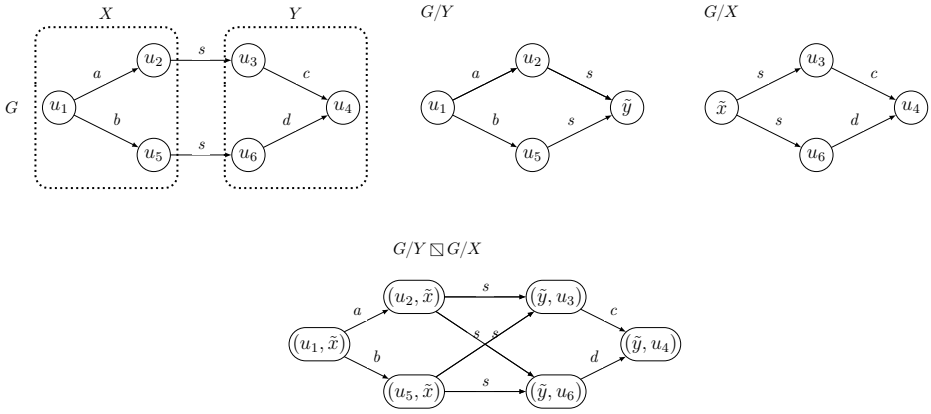


Figure 7.2: Failing decomposition of G into G/X and G/Y, when there exist two different arcs with identical labels in $[X, Y]$.

The next requirement is that the arcs of G/X and G/Y corresponding to the arcs of $[X, Y]$ are the only synchronising arcs of G/X and G/Y. The examples in Figure 7.3 show that this requirement cannot be omitted, without violating the conclusion, if we keep satisfying the other conditions of Theorem 7.1.1.

The case where $G[Y]$ and $[X, Y]$ have arcs with the same label is similar to the example in the right half of Figure 7.3, by symmetry arguments.

Another essential requirement is that $S'(G) \subseteq X$, as the example graph of Figure 7.4 shows. Here, $S'(G) \nsubseteq X$, but X still contains a vertex of $S'(G)$.

For the final requirement that $[X, Y]$ has no backward arcs, the situation is a bit different. In principle, the decomposition would still work fine if all the other conditions are met, but the problem here is that contraction may lead to graphs with directed cycles. So, such situations do not yield useful results in the context of our applications, and formally we did not define the VRSP for such graphs. An example is shown in Figure 7.5. Without giving the details, we observe that the decomposition is valid, i.e., $G \cong G/Y \boxtimes G/X$, but both G/Y and G/X contain directed cycles (of length 2). Note, that the graph G in this figure admits another partition that satisfies all the conditions of Theorem 7.1.1.

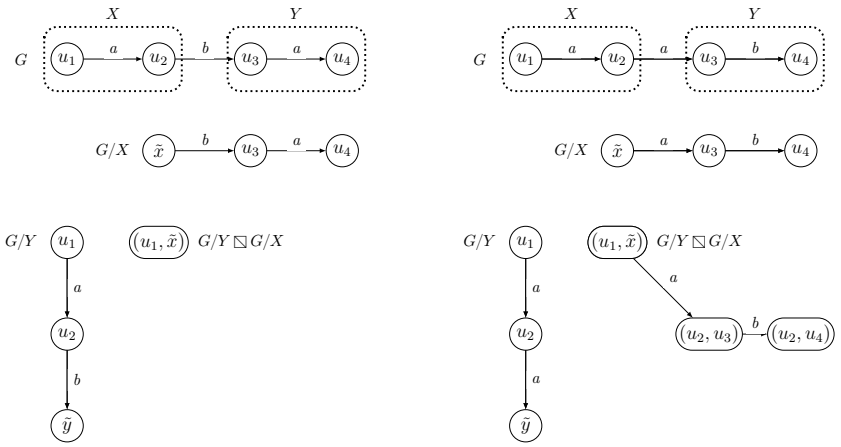


Figure 7.3: Failing decomposition of G into G/X and G/Y, when $G[X]$ and $G[Y]$ (left example) or $G[X]$ and $[X, Y]$ (right example) have arcs with the same label.

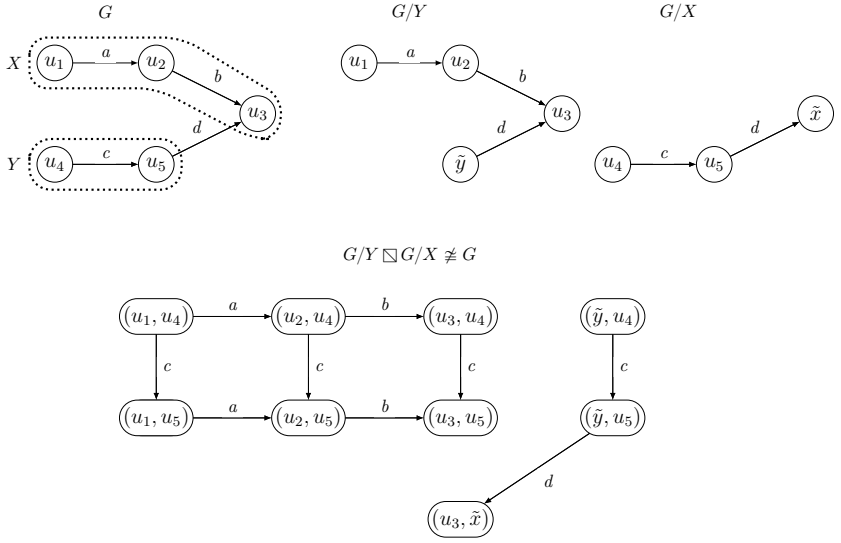


Figure 7.4: Failing decomposition of G in G/Y and G/X, where $S'(G) \nsubseteq X$ and X contains a vertex of $S'(G)$.

In fact, any acyclic directed graph has a partition of its vertex set into sets X and Y such that $[X, Y]$ only contains forward arcs, simply obtained by partitioning the graph according to the levels of its vertices: putting all vertices at $level < j$

in X and all vertices at *level* $\geqslant j$ in Y for a suitable value of j. Clearly, such partitions are easy to find, also algorithmically.

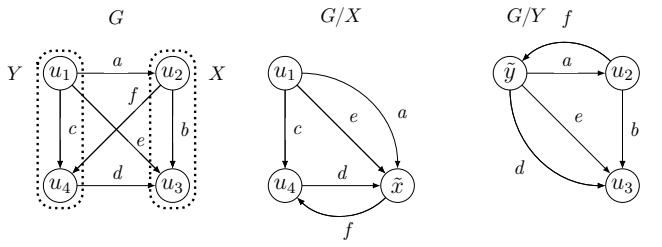


Figure 7.5: Decomposition of G in G/X and G/Y, where $[X, Y]$ contains both backward and forward arcs.

To conclude this section, we add a few remarks on the use of Theorem 7.1.1. Although this theorem has been presented for one (connected) graph G, we recall that the motivation behind the theorem is that the result enables us to replace a graph $G = \sum_{i=1}^{\omega(G)} G_i$ representing $\omega(G)$ processes by a new graph $G' = \sum_{i=1}^{\omega(G')} G'_i$ if at least one of the components G_i of G satisfies the conditions of the theorem. Obviously, Theorem 7.1.1 has quite a few restrictive requirements, so it is rather easy to come up with examples for which the theorem cannot be applied. On the other hand, there are cases in which we can still apply the theorem if the conditions are not met, e.g., if $[X, Y]$ has some backward arcs but G/X and G/Y are acyclic. In the next section, we present an alternative decomposition method for cases in which Theorem 7.1.1 cannot be applied.

7.2 The Second Decomposition Result

In the second decomposition result, we are dealing with a partition of the vertex set into three instead of two nonempty sets. From this alternative partition, we again derive two new graphs: one by subsequently contracting two of the sets of the partition, and one by contracting only the third set. Formally, we define this as follows.

Let X_1, X_2 and Y be three disjoint subsets of $V(G)$, where only X_2 is allowed to be empty, such that $Y = V(G) \backslash (X_1 \cup X_2)$. We use $G/X_1/X_2$ as shorthand for $(G/X_1)/X_2$. In the second decomposition theorem, we give a number of conditions that together guarantee that $G \cong G/Y \boxtimes G/X_1/X_2$. An example of this decomposition in which Y is a vertex cut that separates X_1 and X_2 in G is given in Figure 7.6. A second example, in which Y is not a vertex cut is given in Figure 7.7. As with Theorem 7.1.1, it is not difficult to give examples to show that none of the sufficient conditions can be omitted without violating the conclusion.

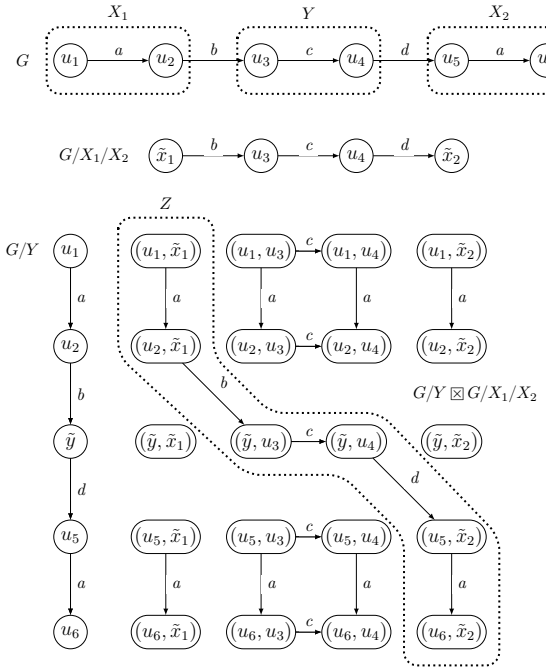


Figure 7.6: Decomposition of G in $G/X_1/X_2$ and G/Y, where Y is a vertex cut that separates X_1 and X_2 in G.

Theorem 7.2.1. *Let G be a graph, and let X_1, X_2 and $Y = V(G)\backslash(X_1 \cup X_2)$ be three disjoint subsets of $V(G)$, where only X_2 is allowed to be empty. Suppose that all the arcs of $[X_1, Y]$ have distinct labels, all the arcs of $[Y, X_2]$ have distinct labels, all the arcs of $[X_1, X_2]$ have distinct labels, the arcs of $[X_1, X_2]$ have no labels in common with any arcs in $[X_1, Y] \cup [Y, X_2]$, and that the arcs of $G/X_1/X_2$ and G/Y corresponding to the arcs of $[X_1, Y] \cup [Y, X_2] \cup [X_1, X_2]$ are the only synchronising arcs of $G/X_1/X_2$ and G/Y. If $S'(G) \subseteq X_1$, and $[X_1, Y]$, $[Y, X_2]$ and $[X_1, X_2]$ have no backward arcs, then $G \cong G/Y \boxtimes G/X_1/X_2$.*

Proof. The proof is analogous to the proof of Theorem 7.1.1. First, we observe that if X_2 is empty, Theorem 7.2.1 is identical to Theorem 7.1.1. Therefore, we assume that X_2 is not empty.

It suffices to define a mapping $\phi : V(G) \to V(G/Y \boxtimes G/X_1/X_2)$ and to prove that ϕ is an isomorphism from G to $G/Y \boxtimes G/X_1/X_2$.

Let $\tilde{x}_1$, $\tilde{x}_2$ and $\tilde{y}$ be the new vertices replacing the sets X_1, X_2 and Y when defining $G/X_1/X_2$ and G/Y, respectively. Consider the mapping $\phi : V(G) \to V(G/Y \boxtimes G/X_1/X_2)$ defined by $\phi(u) = (u, \tilde{x}_1)$ for all $u \in X_1$, $\phi(v) = (v, \tilde{x}_2)$ for all $v \in X_2$ and $\phi(w) = (\tilde{y}, w)$ for all $w \in Y$.

Then ϕ is clearly a bijection if $V(G/Y \boxbslash G/X_1/X_2) = Z$, where Z is defined as $Z = \{(u, \tilde{x}_1) \mid u \in X_1\} \cup \{(v, \tilde{x}_2) \mid v \in X_2\} \cup \{(\tilde{y}, w) \mid w \in Y\}$. We are going to show this later by arguing that all the other vertices of $G/Y \square G/X_1/X_2$ will disappear from $G/Y \boxtimes G/X_1/X_2$. But first we are going to prove the following claim.

Claim 7.2.2. *The subgraph of $G/Y \boxtimes G/X_1/X_2$ induced by Z is isomorphic to G.*

Proof. Obviously, ϕ is a bijection from $V(G)$ to Z. It remains to show that this bijection preserves the arcs and their labels. By the definition of the Cartesian product, for each arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in X_1$ and $v \in X_1$, there exists an arc b in $G/Y \boxtimes G/X_1/X_2$ with $\mu(b) = ((u, \tilde{x}_1), (v, \tilde{x}_1)) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$. Likewise, for each arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in Y$ and $v \in Y$, there exists an arc b in $G/Y \boxtimes G/X_1/X_2$ with $\mu(b) = ((\tilde{y}, u), (\tilde{y}, v)) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$, and for each arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in X_2$ and $v \in X_2$, there exists an arc b in $G/Y \boxtimes G/X_1/X_2$ with $\mu(b) = ((u, \tilde{x}_2), (v, \tilde{x}_2)) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$. Next, first consider an arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in X_1$ and $v \in Y$. For such an arc, in $G/Y \square G/X_1/X_2$ there exist four arcs with label $\lambda(a)$, namely the arcs with $\mu = ((u, \tilde{x}_1), (\tilde{y}, \tilde{x}_1))$, $\mu = ((\tilde{y}, \tilde{x}_1), (\tilde{y}, v))$, $\mu = ((u, \tilde{x}_1), (u, v))$, and $\mu = ((u, v), (\tilde{y}, v))$. In $G/Y \boxtimes G/X_1/X_2$, these four arcs are replaced by one arc b with $\mu(b) = ((u, \tilde{x}_1), (\tilde{y}, v)) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$. Secondly, consider an arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in Y$ and $v \in X_2$. For such an arc, in $G/Y \square G/X_1/X_2$ there also exist four arcs with label $\lambda(a)$, namely the arcs with $\mu = ((\tilde{y}, u), (v, u))$, $\mu = ((v, u), (v, \tilde{x}_2))$, $\mu = ((\tilde{y}, u), (\tilde{y}, \tilde{x}_2))$, and $\mu = ((\tilde{y}, \tilde{x}_2), (v, \tilde{x}_2))$. In $G/Y \boxtimes G/X_1/X_2$, these four arcs are replaced by one arc b with $\mu(b) = ((\tilde{y}, u), (v, \tilde{x}_2)) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$. Thirdly, consider an arc $a \in A(G)$ with $\mu(a) = (u, v)$ for $u \in X_1$ and $v \in X_2$. For such an arc, in $G/Y \square G/X_1/X_2$ there also exist four arcs with label $\lambda(a)$, namely the arcs with $\mu = ((u, \tilde{x}_1), (u, \tilde{x}_2))$, $\mu = ((u, \tilde{x}_1), (v, \tilde{x}_1))$, $\mu = ((v, \tilde{x}_1), (v, \tilde{x}_2))$, and $\mu = ((u, \tilde{x}_2), (v, \tilde{x}_2))$. In $G/Y \boxtimes G/X_1/X_2$, these four arcs are replaced by one arc b with $\mu(b) = ((u, \tilde{x}_1), (v, \tilde{x}_2)) = (\phi(u), \phi(v))$ and $\lambda(b) = \lambda(a)$. Since there are no backward arcs in $[X_1, Y]$, $[Y, X_2]$ and $[X_1, X_2]$, the above arcs are the only arcs in $G/Y \boxtimes G/X_1/X_2$ induced by the vertices of Z. This completes the proof of Claim 7.2.2. □

We continue with the proof of Theorem 7.2.1. It remains to show that all other vertices of $G/Y \boxtimes G/X_1/X_2$, except for the vertices of Z, disappear from $G/Y \boxtimes G/X_1/X_2$. This is clear for the vertex $(\tilde{y}, \tilde{x}_1)$: all the arcs of $G/Y \square G/X_1/X_2$ corresponding to the arcs of $[X_1, Y]$ are synchronising arcs of G/Y and $G/X_1/X_2$, so they disappear from $G/Y \boxtimes G/X_1/X_2$. Hence, $(\tilde{y}, \tilde{x}_1)$ has in-degree 0 in $G/Y \boxtimes G/X_1/X_2$, while it has *level* > 0 in $G/Y \square G/X_1/X_2$. For the other vertices, the argument is as follows.

The vertex set of $G/Y \square G/X_1/X_2$ consists of the union of $Z \cup \{(\tilde{y}, \tilde{x}_1), (\tilde{y}, \tilde{x}_2)\}$ and the vertex sets $(X_1 \cup X_2) \times Y$, $X_1 \times \{\tilde{x}_2\}$ and $X_2 \times \{\tilde{x}_1\}$. We will argue that all

vertices of $(X_1 \cup X_2) \times Y$, $X_1 \times \{\tilde{x}_2\}$ and $X_2 \times \{\tilde{x}_1\}$, as well as the vertex $(\tilde{y}, \tilde{x}_2)$ will eventually disappear from $G/Y \boxtimes G/X_1/X_2$.

Firstly, we claim that all $(u, v) \in X_1 \times Y$ have *level* > 0 in $G/Y \square G/X_1/X_2$. This is obvious if u has *level* > 0 in $G[X_1]$ or v has *level* > 0 in $G[Y]$. Now let $(u, v) \in X_1 \times Y$ such that u has *level* 0 in $G[X_1]$ and v has *level* 0 in $G[Y]$. Then the claim follows from the fact that v has at least one in-arc from a vertex in X_1, since $S'(G) \subseteq X_1$. In fact, since v has only in-arcs from vertices in X_1 and u has no in-arcs at all, (u, v) has *level* 0 in $G/Y \boxtimes G/X_1/X_2$. Hence, all vertices $(u, v) \in X_1 \times Y$ such that u has *level* 0 in $G[X_1]$ and v has *level* 0 in $G[Y]$ disappear from $G/Y \boxtimes G/X_1/X_2$, together with all the arcs with tail (u, v) for all such vertices $(u, v) \in X_1 \times Y$. If after this first step there are still vertices of $X_1 \times Y$ left in $G/Y \boxtimes G/X_1/X_2$, we can repeat the above arguments step by step for such remaining vertices $(u, v) \in X_1 \times Y$ for which (u, v) has the lowest level in what has remained from $G/Y \boxtimes G/X_1/X_2$. Since $G/Y \boxtimes G/X_1/X_2$ is acyclic, it is clear that all vertices of $X_1 \times Y$ disappear one by one from $G/Y \boxtimes G/X_1/X_2$. Now, since $(\tilde{y}, \tilde{x}_2)$ has possibly only in-arcs from vertices $(u, v) \in X_1 \times Y$, $(\tilde{y}, \tilde{x}_2)$ will disappear as well.

Next, we claim that all $(u, v) \in X_2 \times Y$ have *level* > 0 in $G/Y \square G/X_1/X_2$. This is obvious if u has *level* > 0 in $G[X_2]$ or v has *level* > 0 in $G[Y]$. Now let $(u, v) \in X_2 \times Y$ such that u has *level* 0 in $G[X_2]$ and v has *level* 0 in $G[Y]$. Then the claim follows from the fact that u has at least one in-arc from a vertex in Y, since $[Y, X_2]$ has only forward arcs. In fact, since u has only in-arcs from vertices in Y and v has no in-arcs at all, (u, v) has *level* 0 in $G/Y \boxtimes G/X_1/X_2$. Hence, all vertices $(u, v) \in X_2 \times Y$ such that u has *level* 0 in $G[X_2]$ and v has *level* 0 in $G[Y]$ disappear from $G/Y \boxtimes G/X_1/X_2$, together with all the arcs with tail (u, v) for all such vertices $(u, v) \in X_2 \times T$. If after this first step there are still vertices of $X_2 \times Y$ left in $G/Y \boxtimes G/X_1/X_2$, we can repeat the above arguments step by step for such remaining vertices $(u, v) \in X_2 \times Y$ for which (u, v) has the lowest level in what has remained from $G/Y \boxtimes G/X_1/X_2$. Since $G/Y \boxtimes G/X_1/X_2$ is acyclic, it is clear that all vertices of $X_2 \times Y$ disappear one by one from $G/Y \boxtimes G/X_1/X_2$.

We continue with the claim that all $(u, \tilde{x}_1) \in X_2 \times \{\tilde{x}_1\}$ have *level* > 0 in $G/Y \square G/X_1/X_2$. This is obvious if u has *level* > 0 in $G[X_2]$. Now let $(u, \tilde{x}_1) \in X_2 \times \{\tilde{x}_1\}$ such that u has *level* 0 in $G[X_2]$. Then the claim follows from the fact that u has at least one in-arc from a vertex in Y, since $[Y, X_2]$ has only forward arcs. In fact, since u has only in-arcs from vertices in Y and $\tilde{x}_1$ has no in-arcs at all, $(u, \tilde{x}_1)$ has *level* 0 in $G/Y \boxtimes G/X_1/X_2$. Hence, all vertices $(u, \tilde{x}_1) \in X_2 \times \{\tilde{x}_1\}$ such that u has *level* 0 in $G[X_2]$ disappear from $G/Y \boxtimes G/X_1/X_2$, together with all the arcs with tail $(u, \tilde{x}_1)$ for all such vertices $(u, \tilde{x}_1) \in X_2 \times \{\tilde{x}_1\}$. If after this first step there are still vertices of $X_2 \times \{\tilde{x}_1\}$ left in $G/Y \boxtimes G/X_1/X_2$, we can repeat the above arguments step by step for such remaining vertices $(u, \tilde{x}_1) \in X_2 \times \{\tilde{x}_1\}$ for which $(u, \tilde{x}_1)$ has the lowest level in what has remained from $G/Y \boxtimes G/X_1/X_2$. Since $G/Y \boxtimes G/X_1/X_2$ is acyclic, it is clear that all vertices of $X_2 \times \{\tilde{x}_1\}$ disappear one by one from $G/Y \boxtimes G/X_1/X_2$.

Finally, we claim that all $(u, \tilde{x}_2) \in X_1 \times \{\tilde{x}_2\}$ have $level > 0$ in $G/Y \square G/X_1/X_2$. This is obvious if u has $level > 0$ in $G[X_1]$. Now let $(u, \tilde{x}_2) \in X_1 \times \{\tilde{x}_2\}$ such that u has $level\,0$ in $G[X_1]$. Then the claim follows from the fact that $\tilde{x}_2$ has at least one in-arc from a vertex in Y, since $[Y, X_2]$ has only forward arcs. Noting that $\tilde{x}_2$ has only in-arcs from vertices in Y, and all $u \in S'(G) \subseteq X_1$ have no in-arcs at all, clearly for all $u \in S'(G) \subseteq X_1$, $(u, \tilde{x}_2)$ has $level\,0$ in $G/Y \boxtimes G/X_1/X_2$. Hence, all vertices $(u, \tilde{x}_2) \in X_1 \times \{\tilde{x}_2\}$ such that u has $level\,0$ in $G[X_1]$ disappear from $G/Y \boxtimes G/X_1/X_2$, together with all the arcs with tail $(u, \tilde{x}_2)$ for all such vertices $(u, \tilde{x}_2) \in X_1 \times \{\tilde{x}_2\}$.

If after this first step there are still vertices of $X_1 \times \{\tilde{x}_2\}$ left in $G/Y \boxtimes G/X_1/X_2$, we can repeat the above arguments step by step for such remaining vertices $(u, \tilde{x}_2) \in X_1 \times \{\tilde{x}_2\}$ for which $(u, \tilde{x}_2)$ has the lowest level in what has remained from $G/Y \boxtimes G/X_1/X_2$. Since $G/Y \boxtimes G/X_1/X_2$ is acyclic, it is clear that all vertices of $X_1 \times \{\tilde{x}_2\}$ disappear one by one from $G/Y \boxtimes G/X_1/X_2$.

This completes the proof of Theorem 7.2.1. □

We are next going to provide some examples to show that none of the essential conditions in Theorem 7.2.1 can be omitted without violating the conclusion. First of all, it is clear that we need a proper partition of $V(G)$ into nonempty sets X_1, Y and a possibly empty set X_2; otherwise, the contractions cannot be carried out and the whole discussion is meaningless. In fact, the result is only meaningful if at least the partite sets X_1 and Y have at least two vertices. But, note that if we would allow X_1 to be empty instead of X_2, Theorem 7.2.1 is also identical to Theorem 7.1.1. Then we replace Y in Theorem 7.2.1 by X and X_2 in Theorem 7.2.1 by Y. For the other conditions, we show by small examples that they are essential for the validity of the conclusion.

One of the requirements is that the arcs of $[X_1, Y], [Y, X_2]$ and $[X_1, X_2]$ have distinct labels, and $L([X_1, X_2]) \cap L([X_1, Y] \cup [Y, X_2]) = \varnothing$. The example in Figure 7.8 clearly shows that we cannot omit this requirement. Note that all the other conditions of Theorem 7.1.1 are met by this example graph.

The next requirement is that the arcs of $G/X_1/X_2$ and G/Y corresponding to the arcs of $[X_1, Y] \cup [Y, X_2] \cup [X_1, X_2]$ are the only synchronising arcs of $G/X_1/X_2$ and G/Y . The examples in Figure 7.9 show that this requirement cannot be omitted, without violating the conclusion, if we keep satisfying the other conditions of Theorem 7.2.1.

The case where only $G[Y]$ and $[X_1, Y]$, or $G[Y]$ and $[Y, X_2]$ have arcs with the same label is similar to the example in the right half (where $(u_1, \tilde{x}_1$ has no out-arcs) respectively the left half (where $(u_6, \tilde{x}_2$ has no out-arcs) of Figure 7.3, by symmetry arguments.

Another essential requirement is that $S'(G) \subseteq X_1$, as the example graph of Figure 7.10 shows. Here, $S'(G) \nsubseteq X$, but X_1 still contains a vertex of $S'(G)$.

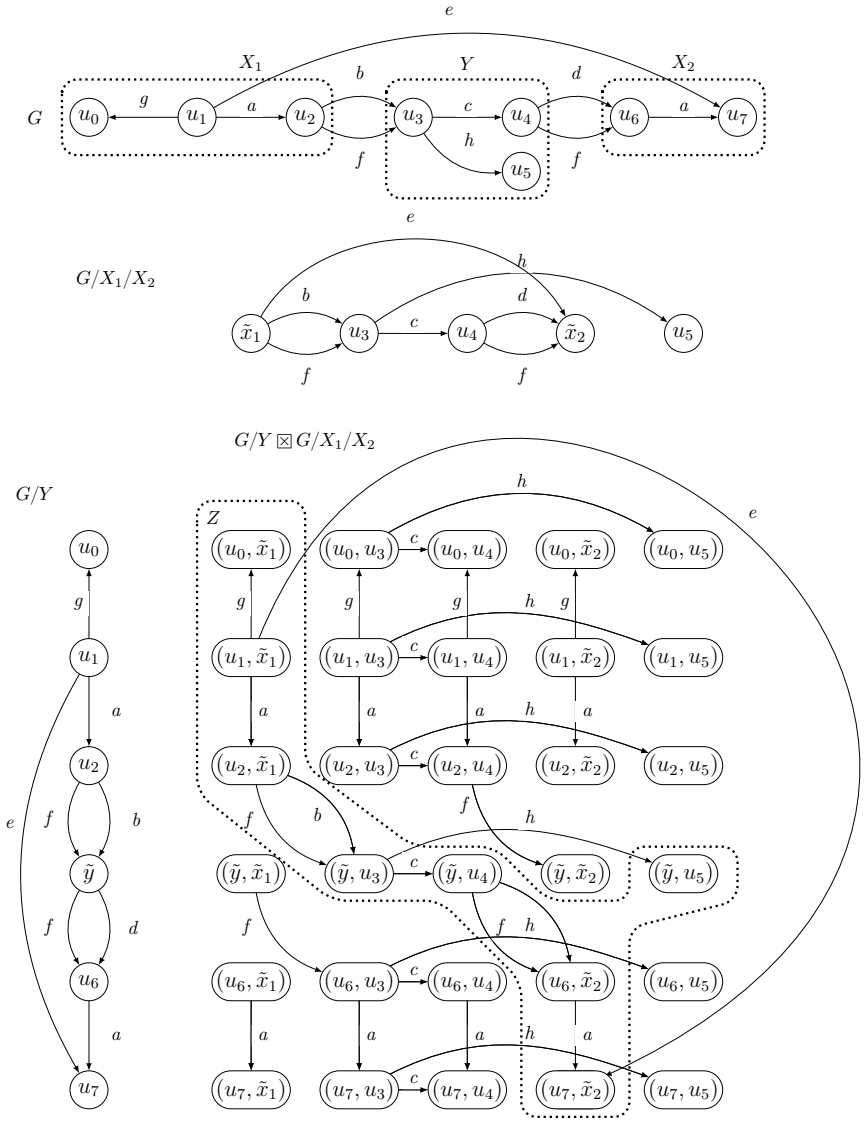


Figure 7.7: Decomposition of G in $G/X_1/X_2$ and G/Y, where Y does not separate X_1 and X_2 in G.

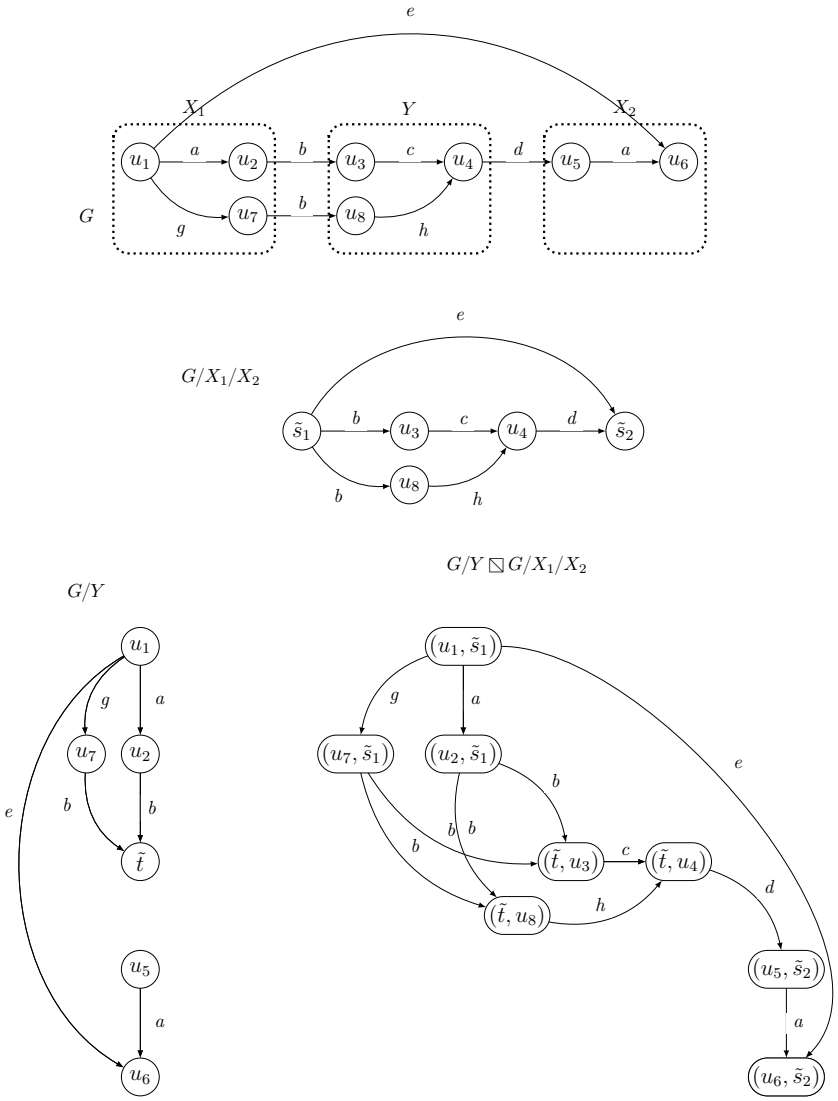


Figure 7.8: Failing decomposition of G in $G/X_1/X_2$ and G/Y, where $[X_1, Y]$ has arcs with identical labels.

For the final requirement that $[X_1, Y], [Y, X_2]$ and $[X_1, X_2]$ have no backward arcs, the situation is a bit different. In principle, the decomposition would still work fine if all the other conditions are met, but the problem here is that contraction may lead to graphs with directed cycles.

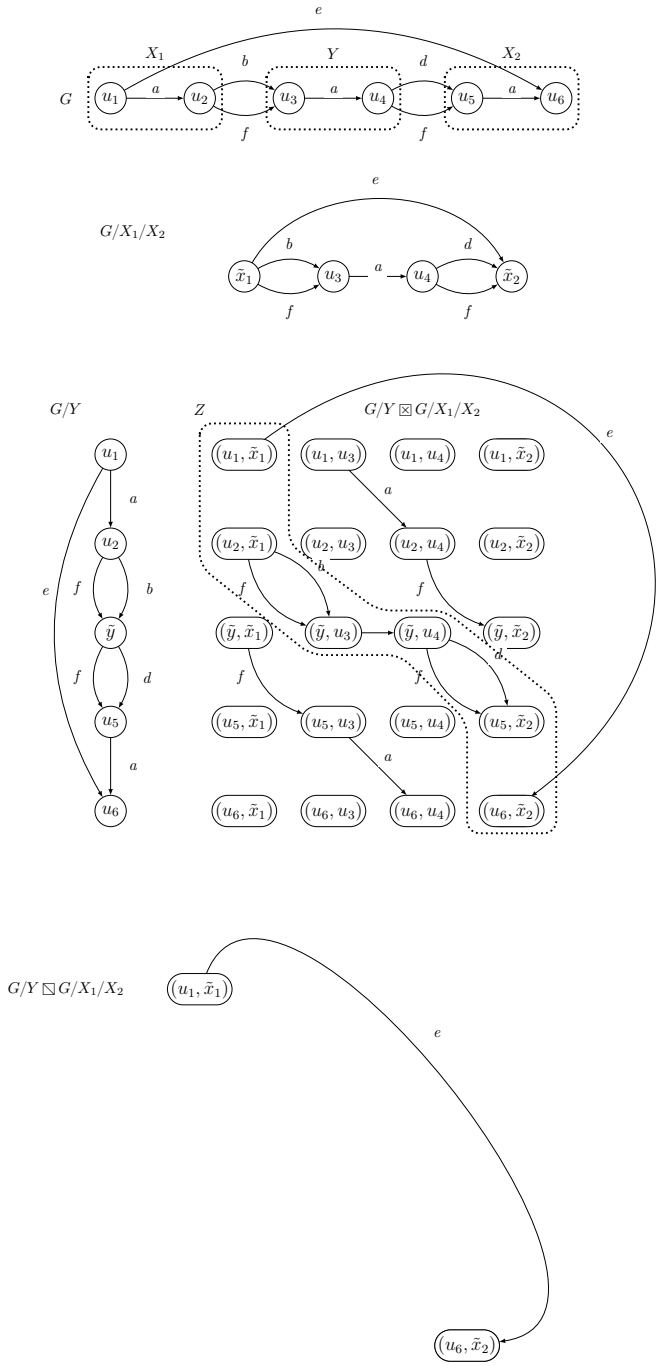


Figure 7.9: Failing decomposition of G in $G/X_1/X_2$ and G/Y, where the arcs of $G/X_1/X_2$ and G/Y corresponding to the arcs of $[X_1, Y] \cup [Y, X_2] \cup [X_1, X_2]$ are not the only synchronising arcs of $G/X_1/X_2$ and G/Y.

So, such situations do not yield useful results in the context of our applications, and formally we did not define the VRSP for such graphs. An example where $[X_1, X_2]$ contains backward arcs is shown in Figure 7.11. The examples where $[X_1, Y]$ and $[Y, X_2]$ contain backward arcs are similar to the example given in Figure 7.11. Without giving the details, we observe that the decomposition is valid, i.e., $G \cong G/Y \boxtimes G/X_1/X_2$, but $G/X_1/X_2$ contains three directed cycles.

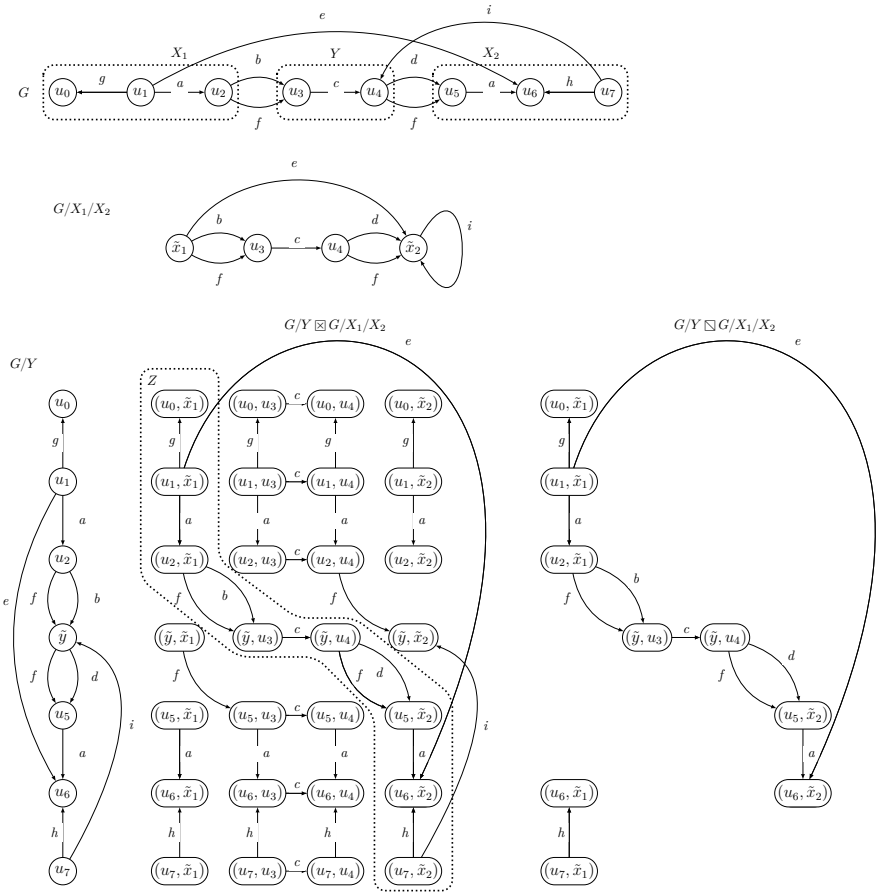


Figure 7.10: Failing decomposition of G in $G/X_1/X_2$ and G/Y, where $X_2 \cap S'(G) \neq \varnothing$.

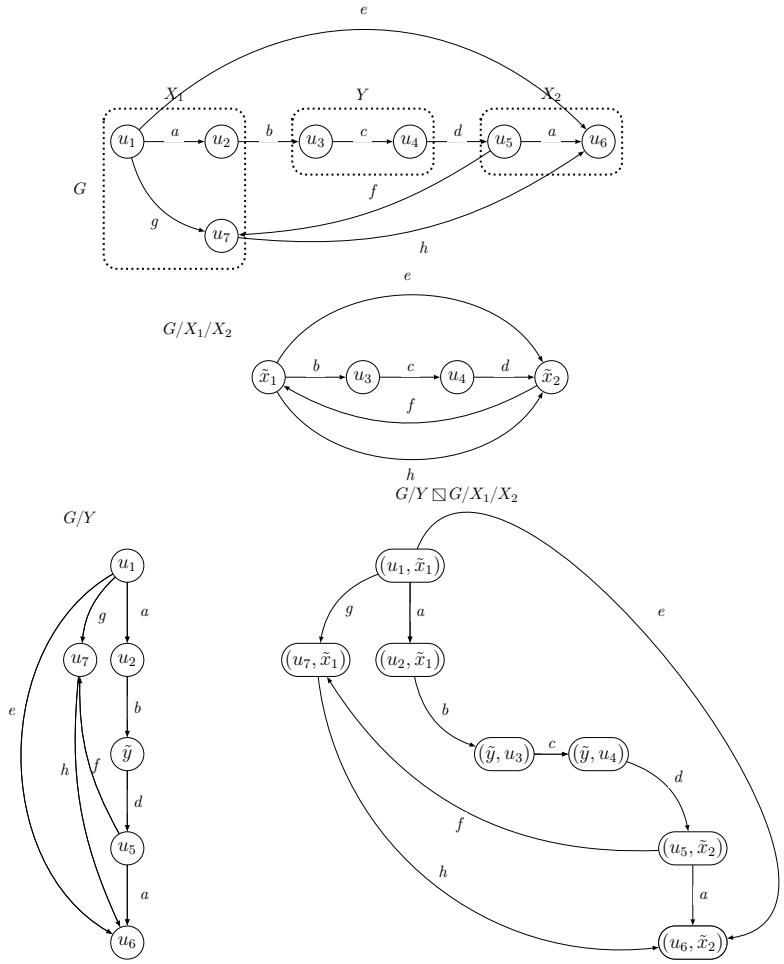


Figure 7.11: Failing decomposition of G in $G/X_1/X_2$ and G/Y, where $[X_1, X_2]$ has backward arcs.

7.3 Applications for Undirected Graphs

We developed the decomposition tools of the previous sections for labelled acyclic directed multigraphs, since these graphs appeared as natural models for the processes and actions in the application area of robotics. In this section, we will show how the tools can be applied to decomposing undirected graphs.

The idea is simple. Let $G = (V, E)$ be a connected undirected (simple or multi)graph. We can orient G, i.e., give directions to the edges of E (repla-

cing each edge $uv \in E$ by one arc a with $\mu(a) = (u, v)$ or $\mu(a) = (v, u)$), in such a way that the resulting directed graph is acyclic. This is a well-known fact. One way to do this, is by just starting at an arbitrary vertex $v \in V$, replacing all edges incident with v by out-arcs of v, considering the graph $G - v$, and repeating this procedure until no edges are left in the remaining graph.

Once we have this connected directed acyclic graph D, we can use any of its suitable arc cuts for our purpose of defining a decomposition. One of the obvious choices would be an arc cut $[X, Y]$ consisting of the arcs between vertices at *level* $\leqslant j$ and *level* $\geqslant j + 1$ for a suitable choice of j, i.e., a choice such that $|X| \geqslant 2$ and $|Y| \geqslant 2$. But many other options are possible, in general. It is not difficult to see that any connected graph on at least four vertices admits such a decomposition. If we then assign different labels to all the arcs of D, the arcs in $[X, Y]$ are the only synchronising arcs of D/X and D/Y, and all the conditions of Theorem 7.1.1 are satisfied. This means that D can be decomposed in D/X and D/Y. This decomposition implies a decomposition for the associated undirected graph G. Such decompositions might turn out to be useful, e.g., as an ingredient of induction proofs within structural graph theory or of recursive methods within algorithmic graph theory. We have no concrete examples to illustrate this.

7.4 Conclusions

In this chapter, we have shown that we can decompose a graph into smaller graphs in such a manner that the VRSP of the decomposed graphs is isomorphic to the original graph. This has led to two theorems, of which the second is a generalisation of the first. In general, these decompositions are not prime decompositions. As an example, if a graph is the Cartesian product of two graph G_1 and G_2, our two theorems do not decompose the product $G_1 \square G_2$ into factors G_1' and G_2' isomorphic to the two graphs G_1 and G_2. Therefore, theory has to be developed similar to the seminal paper of Sabidussi (1960) on the decomposition of graphs by the Cartesian product.

8

Asynchronous Readers and Writers

This chapter is based on our papers presented at the CPA 2016 conference (Boode and Broenink, 2016) and the CPA 2017 conference (Boode and Broenink, 2017).

In PHRCSs, where the control software is designed using process algebras like CSP, information is communicated in a synchronous manner. Even actions in CSP like $c!x : T$ and $c?x : T$ that are interpreted as writing a value x to a channel c and reading a value x from a channel c are just a convenience for $c.x$ on both the writing and the reading side (Schneider, 1999). Therefore, the transfer of data is achieved by synchronisation. Hence, from a process-algebraic point of view, there is no time delay for data transmission between a writing and a reading process, because the data is already available in both the writing and the reading process. Of course, an implementation of these writing and reading actions can be different, e.g. in the sense that the data is sent by a message via a channel from one thread to the other.

Often it is necessary to disconnect the synchronous writing and reading in time; for instance, if a hard real-time thread (e.g. a control loop) has to write a value to a channel, it should not be delayed by a soft real-time thread (e.g. a sequencer) that has to read this value from the channel; otherwise the hard real-time process may miss its deadline. Such an issue can be solved by using a buffer in the CSP-model where a writing process stores its data and the reading process eventually reads the data[1].

Modelling a buffer during the design process in such a manner that it performs its task in the target system without error is arguably not trivial and error-prone. To avoid the use of a buffer process in the CSP model, we propose a new process-algebraic operator for CSP that separates the writing and reading in time. The advantages of such a process-algebraic operator are threefold. Firstly, disconnecting the writing and reading in time by means of this process-algebraic operator at design level (and thereby obsoleting the use of a buffer at design level) eases the task of a designer if such disconnections are required from the

[1]The writing to and reading from a buffer process is performed by synchronising actions between the writing process and the buffer process and synchronising actions between the buffer process and the reading process.

perspective of performance of the application. Secondly, when the buffer is a part of the model[2] it will appear like all other processes in the target system as a thread. To transfer data from the writer thread to the reader thread requires context-switches from the writer thread to the buffer thread and context-switches from the buffer thread to the reader thread. But, if the buffer is a consequence of a process-algebraic operator, it can, for example, be implemented as a shared memory thereby avoiding the buffer-related context-switches. Thirdly, instead of adapting the model of a buffer process for every new application, the buffer mechanism as part of the implementation of a new process-algebraic operator has to be designed and implemented only once.

In line with these disconnections of writing and reading lies a broadcast of data where the sender (writer) process and receiver (reader) process have to be disconnected for performance reasons. In this case, we have one writing process and one or more (synchronous) reading processes. This idea can be extended to one or more (asynchronous) writing processes, and groups of reading processes that read synchronously together with the other reading processes of the same group and read asynchronously with reading processes that are not in the same group.

To achieve these ideas we introduce in this chapter two new CSP operators which disconnect the writing and reading in time.

In Section 8.1 based on Boode and Broenink (2016), we give for CSP a half-synchronous alphabetised parallel operator ${}_{\alpha}\Downarrow_{\beta}$ with alphabets α, β, together with half-synchronous actions $c¡x : T$ and $c¿x : T$ that lies in between synchronous and asynchronous writing and reading; a writer writes asynchronously with respect to a group of readers and the group of readers read synchronously.

We give the syntax and the semantics of the half-synchronous alphabetised parallel operator, together with a case study showing the advantage of this operator with respect to memory occupation and performance.

In Section 8.2 based on Boode and Broenink (2017), we extend the half-synchronous alphabetised parallel operator ${}_{\alpha}\Downarrow_{\beta}$ with alphabets α, β, into the extended half-synchronous alphabetised parallel operator ${}_{\alpha}\Updownarrow_{\beta}$ with alphabets α, β, such that the writers and readers are allowed to write and read asynchronously (one or more (asynchronous) writers and one or more groups of (synchronous) readers. We achieve this by adding an index to the ¿ symbols such that reading actions with the same index read synchronously and reading actions with a different index read asynchronously. As an example, for asynchronous reading of the processes P_1, P_2 and P_3, the actions $c¡x : T \in P_1, c¿x : T \in P_2$ and $c¿x : T \in P_3$ become $c¡x : T \in P_1, c¿_1 x : T \in P_2$ and $c¿_2 x : T \in P_3$. Note that we allow more than one process to write to the same channel. Allowing only one process to write to a channel is a restriction from the early versions of CSP (Hoare, 1978), but lifted to any-to-any channel in, for example, Welch and Martin (2000).

We give the syntax and the semantics of the extended half-synchronous alphabetised

[2]The model according to our system architecture given in Figure 2.1 on page 13.

parallel operator, together with a case study showing the advantage of the extended half-synchronous alphabetised parallel operator with respect to memory occupation and performance.

We finish in section 8.3 with a discussion and a conclusion on the advantages of the (extended) half-synchronous operator.

8.1 The Half-Synchronous Operator

One of the problems that a designer may encounter, is the situation where a process has to communicate a certain value with one or more processes. If this has to be executed synchronously, formal languages like Communicating Sequential Processes (CSP) (Hoare, 1978) supply such a mechanism inherently. But if the actions of writing and reading are asynchronous, languages like CSP have *no* operator that support this. Therefore an arguably complex design has to be made to enforce asynchronous writing and reading.

A mechanism by which the writer and the readers have an optional communication is described by Gruner et al. (2008), called the *optional parallel operator*, denoted as $\Uparrow$. This mechanism is still synchronous in the sense that during the communication the writer and only those readers that are able to receive that data are engaging in the data transfer. All other processes that could receive the data will not engage in the data transfer because they are not in the appropriate state yet. In this manner, the characteristics of synchronous interaction are relaxed to a subset of the reading processes.

Another approach is given by Marwedel (2010) who describes an extended rendezvous, by which the acknowledgement from the receiver to the sender is delayed, such that the receiver can perform checks or calculations on the received data.

We propose a *half-synchronous action* which allows a process to write a value x over a channel c, without the requirement that the reading processes must be in a state where they can read the value x over a channel c. The writing action is denoted as $¡\,(c\,¡\,x : T)$ and the reading action is denoted as $¿\,(c\,¿\,x : T)$. This means that we adjust the alphabetised parallel operator, ${}_X||_Y$, in a similar fashion as Gruner et al. (2008) and introduce the *half-synchronous alphabetised parallel operator* ${}_X\Downarrow_Y$.

For simplicity, we require that the reading processes execute their action $c\,¿\,x : T$ synchronously[3]. In Section 8.2, this requirement is relaxed to a definition of the half-synchronous action, where the writing and reading processes are divided into sets which are set-wise asynchronous, but intra-set-wise synchronous, giving full flexibility to the asynchronous writes and reads.

[3]Like all synchronous actions, this is handled by the Synchronisation Software as described in Chapter 4.

The advantages of the ${}_X\Downarrow_Y$ operator are three-fold;

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

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

8.1.1 Semantics of the Half-Synchronous Operator

In, for example, CSP (Hoare, 1978) one has the possibility to let a process write a value via a variable that will be read by another process using channels. Schneider (1999) describes the communication over a channel as "If c is a channel name of type T, and v is a particular value of type T, then the CSP expression $c\,!v \rightarrow P$ describes a process which is initially willing to output v along channel c, and subsequently behave as P" and "If processes $P(x)$ are defined for each $x \in T$ then the CSP input expression $c\,?x : T \rightarrow P(x)$ describes a process which is initially ready to accept any value x of type T along channel c". But this is still synchronous.

According to Hoare (1978) $c\,!v \rightarrow P_1$ can be written as $c.v \rightarrow P_1$ and $c\,?v \rightarrow P_2$ can be written as $c.v \rightarrow P_2$ where $c.v$ is just an action over which the processes P_1 and P_2 synchronise. Hoare (1978, page 134) observes "the convention that channels are used for communication in only one direction and between only two processes".

For our purpose this communication restricted to two processes in a synchronous manner is too restrictive. Often there is the need for one writer and n readers, for example, in the situation where a process wants to multicast a message to several other processes. It is well known that a designer using, for example, CSP or the Calculus of Communicating Systems (CCS) has sufficient operators to describe any problem at hand (Roscoe, 1998). But such a description may become quite complicated, as an example, if a designer wants to model the observer design pattern (Gamma et al., 1994). To ease the design of concurrent systems an operator supporting such patterns would be convenient from a pragmatic point of view.

Remark 8.1.1. The concept of reading and writing to a buffer is not a rendezvous. An undefined time may elapse between the writing to and the reading from the buffer. Although in a rendezvous there is communication, possibly passing of data between the participating processes, in process algebra a rendezvous is just a synchronising action. If data is passed from one process to another during a rendezvous this is atomic; there is no time elapse between the writing and the reading of the processes.

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

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

As a simple CSP example in Listing 8.1, the processes A, B synchronise over a *sync* action which separate the *write.x* and *read.x* in time. The alphabet of A is X and the alphabet of B is Y.

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

Listing 8.1: Reading from and writing to a buffer.

The graphs G_1, G_2 and $G_1 \boxtimes G_2$ representing the processes A, B and AB are given in Figure 8.1.

Note that Roscoe (2010) gives a more eloquent description of a buffer, which we give in Listing 8.2.

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

Listing 8.2: Reading from and writing to a buffer (Roscoe, 2010).

The optional parallel operator $\Uparrow$, described by Gruner et al. (2008), requires that '*any one or more* of these processes may synchronise with their environment.' It is up to the process whether it will engage in this synchronisation.

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

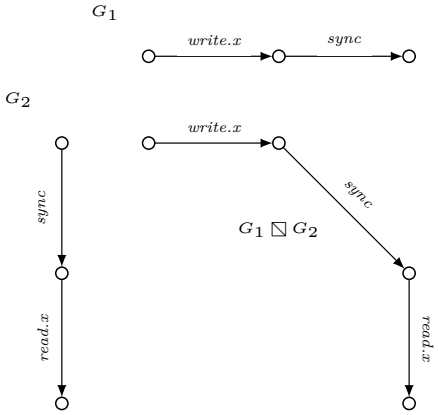


Figure 8.1: G_1, G_2 and $G_1 \boxtimes G_2$.

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

- the *¡-action* is asynchronous and unique with respect to the *¡-actions* of other processes and
- the *¿-action* is enabled if the related *¡-action* (see Definition 8.1.2) has been executed.

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

The rationale is that we want to be able to model one writer and n readers where the waiting-time of the readers is, although timely in a real-time fashion, undefined. In this manner, the writer can continue its task without being delayed by the readers. The readers will read atomically as if in one action. This is where the VRSP shows its strength; the length of the graph is reduced if the processes have the reading of a value on all of their longest paths. The behaviour is closely related to the observer pattern (Gamma et al., 1994)[4].

We use the *big-step relational* semantics described in Nakata and Uustalu (2009) to separate the writing and reading in time. Following Nakata and Uustalu (2009), the proposition $(s, \sigma) \rightarrow (s', \sigma')$ states that in state σ the statement s one-step reduces to s' with the next state being σ'. These are exactly the same as one would use for an inductive semantics, which leads to the terminal many-step reduction relation

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

and allows the possibility of infinitely many steps. The proposition $(s, \sigma) \rightsquigarrow \tau$ expresses that running s from state σ results in the trace τ.

Here we deviate from Nakata and Uustalu (2009). Let $\overset{a}{\rightsquigarrow}$ denote a trace which contains a as an action. Let $\alpha(\rightsquigarrow)$ denote the alphabet containing the actions in $\rightsquigarrow$. Of course, the CSP semantics of an action apply. In Figure 8.2, the relational semantics of the ${}_X\Downarrow_Y$ operator is given, whereby the alphabets of $P, Q_1, \cdots, Q_n, R$ are denoted as $X, Y_1 \cdots, Y_n, Z$, respectively. Furthermore, for alphabets $A_1, A_2, \cdots, A_n$ we define $A_1 \cap A_2 \cap \cdots \cap A_n = (A_1 \cdot A_2 \cdot \ldots \cdot A_n)$ and $A_1 \cup A_2 \cup \cdots \cup A_n = (A_1, A_2, \ldots, A_n)$. For ease of reading we omit in Figure 8.2 for the parallel operator the alphabets, therefore $Q_i \,{}_{Y_i}\Downarrow_{Y_j}\, Q_j$ is denoted as $Q_i \Downarrow Q_j$.

From a graph-theoretical point of view these relational semantics give a restriction on the parallel actions, because a *¿-action* has to wait for the related *¡-action*, as shown in the first rule in Figure 8.2. Therefore, if a full trace τ contains a read, then τ must also contain a related write before the read, hence $c\,¡\,x : T, c\,¿\,x : T \in \tau \Rightarrow c\,¡\,x : T < c\,¿\,x : T$ [5]. The definition of related actions is given in Definition 8.1.2:

Definition 8.1.2. *Two actions are related if and only if*

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

8.1.2 Impact on the VRSP

Of course, the $\Downarrow$ operator leads to an adjustment of the definition of the VRSP ($\boxbslash$) and its intermediate stage ($\boxtimes$) into the DVRSP ($\overset{\bullet}{\boxbslash}$) and its dot intermediate stage ($\overset{\bullet}{\boxtimes}$).

As an example in Figure 8.4 we show the graph representing the case where n values are written by process P_1 and all or none are read by process P_2 [6]. The processes P_1 and P_2 are represented by graphs G_1 and G_2 in Figure 8.3 and Figure 8.4. Because the DVRSP is defined in two stages, we give the dot intermediate stage of G_1, G_2 and $G_1 \overset{\bullet}{\boxtimes} G_2$, in Figure 8.3 and the DVRSP of G_1, G_2 and $G_1 \overset{\bullet}{\boxbslash} G_2$, in Figure 8.4. Note that $G_1 \overset{\bullet}{\boxtimes} G_2$ consists of three components and $G_1 \overset{\bullet}{\boxbslash} G_2$ consists of one component. Two components are removed in the second stage of the DVRSP, because the level of the sources of these components are zero, whereas the level of these vertices in the Cartesian product of G_1, G_2 and $G_1 \square G_2$ are greater than zero.

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

[6] The *waitForNextPeriod* action in Figure 8.4 is defined as a method in the class *RealtimeThread* in the Real-Time Specification for Java (Bollella, 2000; Wellings, 2004).

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

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

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

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

Figure 8.2: Relational semantics of the half-synchronous operator for a specification comprising the processes $P, Q_1, \cdots, Q_n, R$.

Remark 8.1.3. The definition of a label has to be augmented. Boode et al. (2013) have given as a definition for a label "For each arc $a \in A$, $\lambda(a) \in L$ consists of a pair $(l(a), t(a))$, where $l(a)$ is a string representing an action and $t(a)$ is a positive real number representing the worst-case execution time of the action represented by $l(a)$". $l(a)$ is augmented by the restriction that whenever ¿ and ¡ are in $l(a)$, the arc with label $\lambda(a)$ is either representing a reading or writing action.

Remark 8.1.4. Let the processes P_1 and P_2, represented by graphs G_1 and G_2 respectively, half-synchronise over some writing action $c\,¡\,x_i : T$ of P_1 and some reading action $c\,¿\,x_i : T$ of P_2 on some channel c. Then an arc representing a reading action $c\,¿\,x_i : T$ on some channel c, only makes sense in the product $G_1 \dot{\boxtimes} G_2$ if every path from the source of $G_1 \dot{\boxtimes} G_2$ to the arc representing the reading action $c\,¿x_i : T$ contains an arc representing a related writing action $c\,¡x_i : T$.

Therefore, let b be an arc in $A(G_2)$ with $\mu(b) = (u_2, v_2), \lambda(b) = c\,¡\,x_i : T, u_2, v_2 \in V(G_2)$. Then, for an arc a with $\mu(a) = (u_1, v_1)$, $\lambda(a) = c\,¿\,x_i : T$, $u_1, v_1 \in V(G_1)$, there must be an arc b in every path from the source of $G_1 \dot{\boxtimes} G_2$ to the vertex $(u_1, v_2) \in V(G_1 \dot{\boxtimes} G_2)$. Whenever this is not the case, i.e. there is no other process with a writing action $c\,¡x_i : T$, the reading process will encounter a deadlock when the reading action $c\,¿\,x : T$ is the only action the reading process can execute. The opposite, where a writing action $c\,¡x_i : T$ is not followed by a related reading action $c\,¿\,x : T$, is not prohibited. Although seemingly useless, we do not prohibit the writing of values without reading.

For two graphs G_i and G_j, an arc $a \in A_i$ with $\mu(a) = (u_i, v_i)$ is related to an

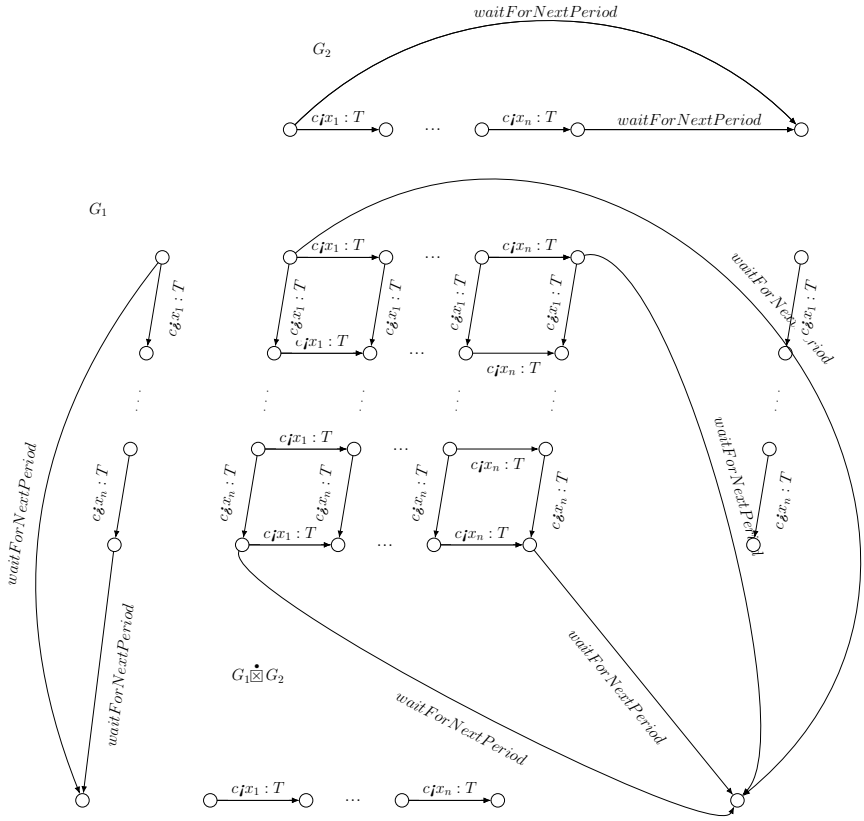


Figure 8.3: The intermediate stage of the DVRSP of G_1 and G_2, $G_1 \dot{\boxtimes} G_2$.

arc $b \in A_j$ with $\mu(b) = (u_j, v_j)$ if in the processes represented by G_i and G_j, the actions they represent are related. Related actions can lead to a form of inconsistency that is not covered by Definition 6.2.1 as is shown in Figure 8.10 on page 117 where there is a path representing a trace with a writing action that is followed by two related reading actions. Therefore we have to incorporate the number of related writing actions and related reading actions in each path from the source to the sink of a graph in Definition 6.2.1. But first, we have to define how we are counting the number of related writing and reading actions in a path, which we will use in the enhanced definition of consistency of graphs, and adapt the definition of the VRSP leading to the DVRSP.

The *path write cardinality* of a path P of a graph G with respect to an arc a in P

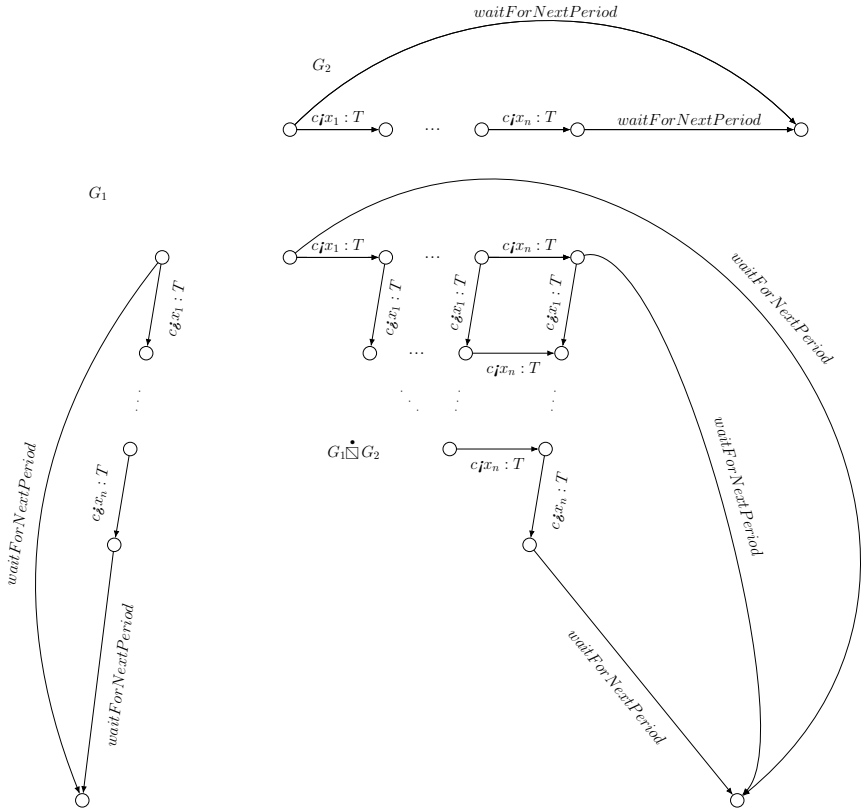


Figure 8.4: Half-synchronous writing and reading of $G_1 \dot{\boxtimes} G_2$ and its factors G_1, G_2.

with $\mu(a) = (u, v)$, $\lambda(a) = c_{\text{¡}}x : T$, denoted as $P(c_{\text{¡}}x : T)$, is defined as the number of occurrences of writing actions $c_{\text{¡}}x : T$ in the path P.

The *path read cardinality* of a path P of a graph G with respect to an arc a in P with $\mu(a) = (u, v), \lambda(a) = c_{\text{¿}}x : T$, denoted as $P(c_{\text{¿}}x : T)$, is defined as the number of occurrences of reading actions $c_{\text{¿}_n}x : T$ in the path P.

The related writing and reading actions also have an impact on the VRSP, therefore we adjust the definition of the VRSP leading to the definition of the DVRSP.

As before, we modify the Cartesian product $G_i \square G_j$ according to the existence of synchronising arcs, but now with the extra constraint that arcs $a \in A(G_1), b \in A(G_2), \lambda(a) = \lambda(b)$ where the labels contain a ¡ character, are removed.

The first step in this modification consists of ignoring the synchronising arcs while forming arcs in the product, but additionally combining pairs of synchronising

arcs of G_i and G_j into one arc, yielding the intermediate product which we denote by $G_i \dot{\boxtimes} G_j$. To be more precise, $G_i \dot{\boxtimes} G_j$ is obtained from $G_i \square G_j$ by first ignoring all except for the so-called asynchronous arcs, i.e., by only maintaining all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_j = w_j$ and $\lambda(a) \notin L_j$, as well as all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_i = w_i$ and $\lambda(a) \notin L_i$.This set of arcs is denoted by $A^a_{i,j}$. Additionally, we add arcs that replace synchronising pairs $a_i \in A_i$ and $a_j \in A_j$ with $\lambda(a_i) = \lambda(a_j)$ and ¡ not in $l(a_j)$. If $\mu(a_i) = (v_i, w_i)$ and $\mu(a_j) = (v_j, w_j)$, such a pair is replaced by an arc $a_{i,j}$ with $\mu(a_{i,j}) = ((v_i, v_j), (w_i, w_j))$ and $\lambda(a_{i,j}) = \lambda(a_i)$ and ¡ not in $l(a_i)$. The set of these so-called synchronous arcs of $G_i \dot{\boxtimes} G_j$ is denoted by $A^s_{i,j}$. Note that synchronous arcs containing a ¡ are not maintained.

Furthermore, let Q be a path from the source of $G_i \dot{\boxtimes} G_j$ to the vertex $(w_i, w_j) \in V_{i,j}$ and let R be any full path of $G_i \dot{\boxtimes} G_j$, then the second step in this modification consists of removing (from $G_i \dot{\boxtimes} G_j$) all arcs a of any path Q with the following condition: a is an arc of Q and $\mu(a) = ((v_i, v_j), (w_i, w_j)), \lambda(a) = l_r, ¿$ in l_r, for which there exists a related arc $b \in A_{i,j}$ with $\mu(b) = ((v_k, v_l), (w_k, w_l)), \lambda(b) = l_w, ¡$ in l_w, and where for Q and all R: $Q(l_r) > Q(l_w)$ and $Q(l_r) \leqslant R(l_w)$. Followed by removing the vertices $(v_i, v_j) \in V_{i,j}$ and the arcs a with $tail(a) = (v_i, v_j)$, whenever (v_i, v_j) has $level > 0$ in $G_i \square G_j$ and (v_i, v_j) has $level\, 0$ in $G_i \dot{\boxtimes} G_j$. This is then repeated in the newly obtained graph, and so on, until there are no more vertices at $level\, 0$ in the current graph that are at $level > 0$ in $G_i \square G_j$.

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

Using the path write cardinality and the path read cardinality, we define the consistency of graphs as follows.

Graphs G_i and G_j are *consistent* if and only if the following three requirements apply:

1. $(\rho_{G_i}(G_i \dot{\boxtimes} G_j))^\delta \cong G_j$ and $(\rho_{G_j}(G_i \dot{\boxtimes} G_j))^\delta \cong G_i$.
2. $S'(G_i \dot{\boxtimes} G_j) = S'(G_i) \times S'(G_j)$ and $S''(G_i \dot{\boxtimes} G_j) = S''(G_i) \times S''(G_j)$.
3. Whenever Q and R are paths from the source to the sink of G_i $(G_j, G_i \dot{\boxtimes} G_j)$, $Q(c_¡ x : T) = R(c_¡ x : T)$ and $Q(c_{¿_k} x : T) = R(c_{¿_k} x : T)$.

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

$$
\begin{aligned}
A' &= c\,¡\,x : T \;\rightarrow\; \mathrm{SKIP} \\
B' &= c\,¿\,x : T \;\rightarrow\; \mathrm{SKIP} \\
A'B' &= A' \,{}_{X}\!\not\Downarrow_{Y'}\, B'
\end{aligned}
$$

Listing 8.3: Reading from and writing to a buffer.

may deadlock[7]. Obviously, if two graphs contain identical writing actions, the processes representing these graphs may deadlock as well.

Remark 8.1.5. Such deadlocks cannot occur for the optional parallel operator because then the effect will be that one of the two processes will not participate in the reading from either x or y.

Remark 8.1.6. The order in which a process reads is not relevant with respect to a process that only writes. For example, if the first process writes to a channel x and then writes to a channel y and the second process reads from a channel y and then reads from a channel x this will not give a deadlock. The result will be that the second process cannot start reading from the channel x before there is written to the channel y.

From a performance point of view, the graph representing the example given in Listing 8.1 has a length of $\ell(G_1 \diagdown\!\!\diagdown G_2) = 3$ [8], whereas for the process representing $G_1 \diagdown\!\!\diagdown G_2$ the same behaviour is achieved by the process $A'B'$ given in Listing 8.3. The length of the graph representing the process $A'B'$ is 2. Although this reduces the number of context switches, the synchronisation software has to deal with the order of execution of the $c\,¡\,x : T$ action and the related $c\,¿\,x : T$ action. Therefore the performance gain depends on the time the synchronisation software needs to control the order of the actions. The alphabet of A' is X' and the alphabet of B' is Y'.

8.1.3 Case Study of the Half-Synchronous Alphabetised Parallel Operator

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

[7] The writing and reading show a close resemblance with databases, where there are transactions writing and reading data concurrently. As an example, Bernstein et al. (1987) show that the order in which data is written and read matters with respect to the consistency (in the sense of interference) of the data. They distinguish three types of execution of transactions; Recoverable executions, Avoiding Cascading Aborts executions and Strict executions. Of course, the updates of the data in database systems have to be committed (the updates are considered valid) or aborted (they are considered as if the updates never happened), which is an aspect of data we do not take into account.

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

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

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

$$
System_1 \;=\; Application_A \,||_C\, Controller
$$

Listing 8.4: Reading from and writing to a buffer.

Assume that the application process has to calculate eight values via the coprocessor. Let the controller process have priority over the application process. Furthermore, the actions of the application process and of the actions of the controller process take 10 μs to execute. This includes context switches, state changes in the processes and the like. The coprocessor takes 70 μseconds to calculate the FFT on each data item. Although the related[9] **!**-actions and **?**-actions communicate as a rendezvous, so in a sense atomically, their interaction takes 20 μseconds. This leads to a simple CSP specification given in Listing 8.4 using **!**-actions and the **?**-actions, where the alphabet of *Application* is A and the alphabet of *Controller* is C.

In Figure 8.5 we show the time line for $System_1$ with the application process (AP), the control process (CP) and the coprocessor (CoP). Obviously, there is a deadline miss because $System_1$ needs more than one ms to execute.

Using the new $\Downarrow$ operator and the *¡-actions* and the *¿-actions*, this leads to an equally simple CSP specification given in Listing 8.5.

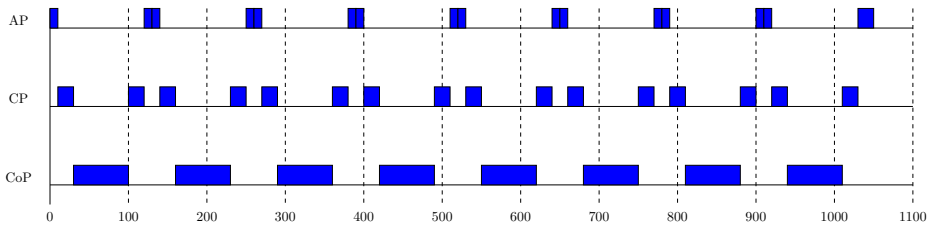


Figure 8.5: Time line of the application process, the control process and the coprocessor, using **!** operator and **?** operator.

In Figure 8.6 we show the time line for $System_2$ with the application process (AP), the control process (CP) and the coprocessor (CoP). Now during the time

[9]Related in a similar fashion as defined for the ¡-actions and ¿-actions in Definition 8.1.2.

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

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

$$
System_2 \;=\; Application \,{}_A\Downarrow_C\, Controller
$$

Listing 8.5: Reading from and writing to a buffer.

that the coprocessor is executing, the application process is writing the $x_2, \cdots, x_8$ values via channel c. Furthermore, the reading of the $y_1, \cdots, y_7$ is as well executed during the execution of the coprocessor. $System_2$ is an improvement of $System_1$ by 140 μseconds as the time line in Figure 8.6 shows.

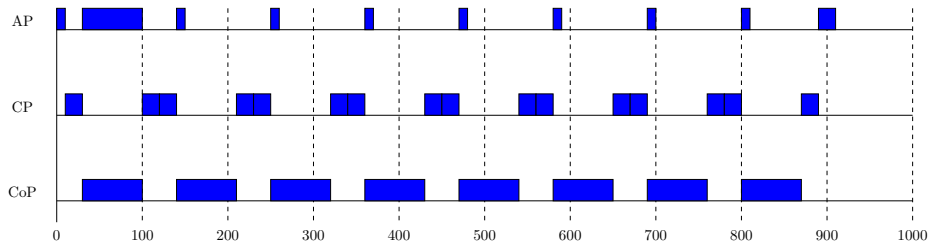


Figure 8.6: Time line of the application process, the control process and the coprocessor, using ¡ operator and ¿ operator.

8.2 Extension of the Half-Synchronous Operator to Asynchronous Readers

Although reading actions are asynchronous for the half-synchronous alphabetised parallel operator, the readers are still synchronising their reading actions. Furthermore, the writers will deadlock if they are trying to invoke the same writing action. In this section, we lift these restrictions such that the readers are allowed to read synchronously as well as asynchronously, and the writers are allowed to write the same value asynchronously.

To achieve this kind of reading by readers, we add an index to the ¿ symbols such that reading actions with the same index read synchronously and reading actions with a different index read asynchronously. For example, for the processes P_1, P_2 and P_3 we have that actions **c¡x:T** of P_1, **c¿x:T** of P_2 and **c¿x:T** of P_3 become

c¡x:T of P_1, **c¿$_1$x:T** of P_2 and **c¿$_2$x:T** of P_3. Furthermore, we allow more than one process to write to the same channel.

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

Remark 8.2.1. Of course, we could index the asynchronous writes in a similar fashion as the asynchronous reads. We choose *not* to, because the writing at any point in time, when delivering *identical* objects to the readers, would lead to the passing of one object only, delaying all threads, but the last, that participate in the synchronisation. This is counter-intuitive to the idea that threads can write on a channel asynchronously, with the guarantee that their instance of an object is written to the channel at that point in time. An adaptation of the optional parallel operator of Gruner et al. for writers would solve this issue.

8.2.1 Semantics of the Extended Half-Synchronous Alphabetised Parallel Operator

Let $P = \{P_1, \cdots, Q\}$ be the set of processes containing an asynchronous $¡ - action$. Let π be a partition $\{I_1, I_2, \ldots, I_s\}$ of the index set $I = \{1, 2, \ldots, m\}$. Let $Q = \{Q_j \mid j \in I_i, I_i \in \pi\}$ be the set of processes containing an indexed asynchronous $¿_i - action$.

Furthermore, in Figure 8.7 we give

- the semantics of the extended half-synchronous operator,
- if we need more than one process P we use P_i; otherwise we use P, and
- the alphabets of $P, P_1, \cdots, R, Q_1, \cdots, Q_n, R$ are denoted as $X, X_1 \cdots, X_m$, $Y_1 \cdots, Y_n, Z$, respectively.

For ease of reading, we omit the alphabets for the extended half-synchronous operator, therefore $Q_i \,{}_{Y_i}\Updownarrow_{Y_j}\, Q_j$ is denoted as $Q_i \Updownarrow Q_j$.

Remark 8.2.2. The **¿$_i$**-action is prone to deadlocks. If one process contains an action $c¿_i x : T$ followed by an action $c¿_j x : T, i \neq j$ and another process contains the same actions in reversed order the two processes may deadlock. Because we consider processes represented by consistent graphs only, such a process specification is inhibited.

8.2.2 The EVRSP of the Extended Half-Synchronous Alphabetised Parallel Operator

As we are taking into account pairs of consistent graphs only, an action $c¿_n x : T$ in one process without an action $c¡x : T$ in any process is inhibited, because

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

$$\frac{P_k \overset{c\,¡\,x:T}{\rightsquigarrow} P_k',\ Q_i \overset{c\,¿_i\,x:T}{\rightarrow} Q_i',\ Q_j \overset{c\,¿_j\,x:T}{\rightarrow} Q_j'}{(P_k \Updownarrow Q_i \Updownarrow Q_j) \overset{c\,¡\,x:T}{\rightsquigarrow} (P_k' \Updownarrow\ Q_i \Updownarrow Q_j) \rightarrow ((P_k' \Updownarrow\ Q_i' \Updownarrow Q_j) \oplus (P_k' \Updownarrow Q_i \Updownarrow Q_j')) \rightarrow (P_k' \Updownarrow Q_i' \Updownarrow Q_j')}, \quad i \neq j,$$

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

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

$$\frac{Q_i \overset{c\,¿_i\,x:T}{\rightarrow} Q_i',\ Q_j \overset{y}{\rightarrow} Q_j'}{Q_i \Updownarrow Q_j \rightarrow Q_i \Updownarrow Q_j'}, y \neq c\,¿_i\,x : T, c\,¿_i\,x : T \in Y_j\}$$

Figure 8.7: Relational semantics of the extended half-synchronous operator for a specification comprising the processes $P, Q_1, \cdots, Q_n, R$.

the process may end in a deadlock and the deadlock violates the consistency requirements. But we still have to address issues like

- a series of identical writing actions $c¡x : T$ in one process and the related reading actions $c¿_n x : T$ in another process,
- a series of consecutive identical writing actions $c¡x : T$ to the same channel by different processes.

These issues are not inhibited by the semantics of ${}_\alpha\Updownarrow_\beta$. As an example, the processes P_1, P_2, P_3 and P_{123} in Listing 8.6 are represented by the graph in Figure 8.9, which contains consistent components G_1, G_2, G_3 leading to $G_{123} = \overset{3}{\underset{i=1}{\overset{\diamond}{\boxtimes}}} G_i$ (the EVRSP, denoted as $\overset{\diamond}{\boxtimes}$, is defined in the sequel). A schema of the processes given in Listing 8.6, is given in Figure 8.8. This process schema describes the communication flow of the involved processes and shows that there is no predefined order in which P_1 and P_2 communicate with P_3. It follows that the components representing these processes must be G_1, G_2, G_{13}'[10] and G_{123}, given in Figure 8.9,

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

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

Listing 8.6: Two processes writing via the same channel.

because $G''_{13} \overset{\diamond}{\boxtimes} G_2 \ncong G_{123}$. Therefore, it is clear that component G_{123} in Figure 8.9 represents the behaviour of the concurrent process P_{123}. But it is not clear what the component should be that represents the concurrent process P_{13} given in Listing 8.6, because the writing action could be related to the first reading action or to the second reading action.

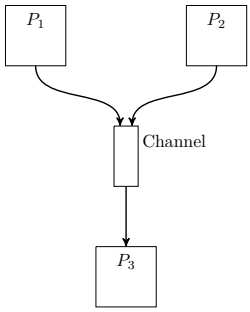


Figure 8.8: Process schema describing the communication flow of the processes P_1, P_2, P_3 (Listing 8.6).

Therefore, there are two choices for this example given by the components G'_{13} and G''_{13} in Figure 8.9. Following the process sketch in Figure 8.8, the component G'_{13} should be chosen because the first two actions can be executed directly by the processes that represent these components, whereas the process representing component G''_{13} has to wait for the writing action $c¡x : T$ of the process representing component G_2.

This problem becomes even worse if we consider the processes given in Listing 8.7. The graph representing the processes of Listing 8.7 contains a path represented by the trace $doX_1 \rightarrow c\,¡\,x : T \rightarrow doY_2 \rightarrow c\,¿_1\,x : T \rightarrow c\,¿_1\,x : T \rightarrow$ SKIP (the thick and dotted arrows in Figure 8.10 [11]), which is obviously wrong. But the dashed and dotted arrows in Figure 8.10 represent a trace that has to be possible. The problem lies in the black vertex in Figure 8.10, that allows two traces to be possible with a different number of writing actions.

[11] For ease of reading the not-relevant labels are removed in Figure 8.10.

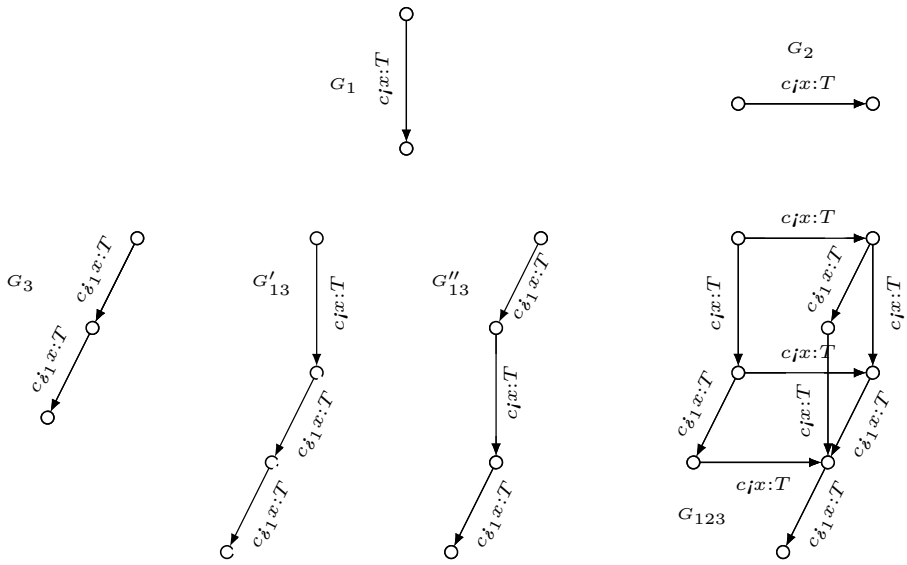


Figure 8.9: Components $G_1, G_2, G_3, G_{123} = \overset{3}{\underset{i=1}{\diamondsuit}} G_i$, and $G'_{13} = G_1 \diamondsuit G_3$ or $G''_{13} = G_1 \diamondsuit G_3$ representing processes P_1, P_2, P_3, P_{123} and P_{13} (Listing 8.6).

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

Listing 8.7: Ambiguity of a writing process and a reading process via the same channel.

Because the components G_1 and G_2 are consistent according to Definition 6.2.1, we have to adjust Definition 6.2.1 incorporating the number of writes and reads in each path from the source to the sink of a component.

Obviously, the components representing the processes in Listing 8.7 are not consistent. But the processes in Listing 8.6 are consistent and therefore the EVRSP has to determine the order of the reading actions with respect to the writing actions.

For the EVRSP whenever two processes contain identical ¡-actions, these actions are treated asynchronously. For indexed ¿-actions, the index makes the ¿-actions different and therefore the EVRSP handles these actions identical to the

DVRSP. Hence, theVRSP must be extended to handle the ¡-actions for components representing different processes only.

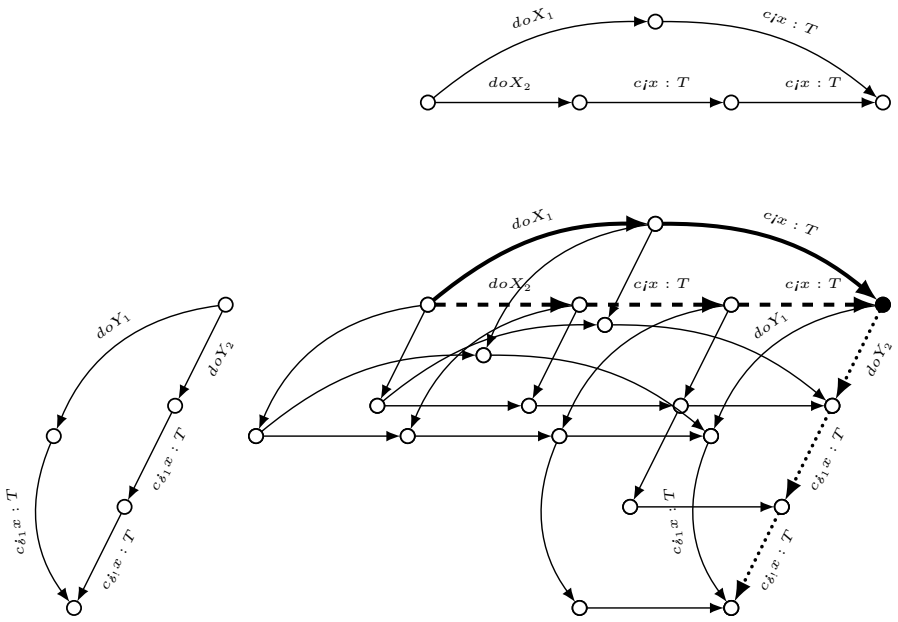


Figure 8.10: Components G_1, G_2 and $G_{12} = \overset{2}{\underset{i=1}{\overset{\diamond}{\boxtimes}}} G_i$ representing processes P_1, P_2 and P_{12} of Listing 8.7.

But first, we have to adapt the definition of the VRSP leading to the definition of the EVRSP, after which, we can define the definition of consistency of graphs for the EVRSP.

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

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

Again, we modify the Cartesian product $G_i \square G_j$ according to the existence of synchronising arcs, but now with the extra constraint that labels containing a ¡

character are asynchronous i.e., pairs of arcs with the same label pair without a ¡ character, with one arc in G_i and one arc in G_j.

The first step in this modification consists of ignoring the synchronising arcs while forming arcs in the product, but additionally combining pairs of synchronising arcs of G_i and G_j into one arc, yielding the intermediate product which we denote by $G_i \overset{\diamond}{\boxtimes} G_j$. To be more precise, $G_i \overset{\diamond}{\boxtimes} G_j$ is obtained from $G_i \square G_j$ by first ignoring all except for the so-called asynchronous arcs, i.e., by only maintaining all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_j = w_j$ and $\lambda(a) \notin L_j$ or $v_j = w_j$ and $\lambda(a) \in L_j$ and ¡ in $l(a)$, as well as all arcs $a \in A_{i,j}$ for which $\mu(a) = ((v_i, v_j), (w_i, w_j))$, whenever $v_i = w_i$ and $\lambda(a) \notin L_i$ or $v_i = w_i$ and $\lambda(a) \in L_i$ and ¡ in $l(a)$. This set of arcs is denoted by $A^a_{i,j}$. Additionally, we add arcs that replace synchronising pairs $a_i \in A_i$ and $a_j \in A_j$ with $\lambda(a_i) = \lambda(a_j)$ and ¡ not in $l(a_j)$. If $\mu(a_i) = (v_i, w_i)$ and $\mu(a_j) = (v_j, w_j)$, such a pair is replaced by an arc $a_{i,j}$ with $\mu(a_{i,j}) = ((v_i, v_j), (w_i, w_j))$ and $\lambda(a_{i,j}) = \lambda(a_i)$ and ¡ not in $l(a_i)$. The set of these so-called synchronous arcs of $G_i \overset{\diamond}{\boxtimes} G_j$ is denoted by $A^s_{i,j}$.

The second step in this modification consists of removing (from $G_i \overset{\diamond}{\boxtimes} G_j$) the vertices $(v_i, v_j) \in V_{i,j}$ and the arcs a with $tail(a) = (v_i, v_j)$, whenever (v_i, v_j) has $level > 0$ in $G_i \square G_j$ and (v_i, v_j) has $level\, 0$ in $G_i \overset{\diamond}{\boxtimes} G_j$ and all arcs $v_x w_x \in A_{i,j}$ for which there exists a related arc $v_y w_y \in A_{i,j}$, with operator ¿$_n$ in $l(v_x w_x)$ for which there does not exist at least n related arcs $v_y w_y$ with operator ¡ in $l(v_y w_y)$ with $v_y w_y < v_x w_x$. This is then repeated in the newly obtained graph, and so on, until there are no more vertices at $level\, 0$ in the current graph that are at $level > 0$ in $G_i \square G_j$.

The resulting graph is called the EVRSP of G_i and G_j, denoted as $G_i \overset{\diamond}{\boxtimes} G_j$.

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

Remark 8.2.3. Because arcs $v_i w_i$ with ¿ $\in l(v_i w_i)$ are indexed, the arcs $v_i w_i$ with labels that differ only by their indices represent asynchronous actions.

Remark 8.2.4. The EVRSP allows two or more processes to write a value to the same channel.

Components G_i and G_j are *consistent* if and only if the following three requirements apply:

1. $(\rho_{G_i}(G_i \overset{\diamond}{\boxtimes} G_j))^\delta \cong G_j$ and $(\rho_{G_j}(G_i \overset{\diamond}{\boxtimes} G_j))^\delta \cong G_i$.
2. $S'(G_i \overset{\diamond}{\boxtimes} G_j) = S'(G_i) \times S'(G_j)$ and $S''(G_i \overset{\diamond}{\boxtimes} G_j) = S''(G_i) \times S''(G_j)$.
3. Whenever Q and R are paths from the source to the sink of G_i $(G_j, G_i \overset{\diamond}{\boxtimes} G_j)$, $Q(c$¡x:T$) = R(c¡x : T)$ and $Q(c$¿$_k$x:T$) = R(c¿_k x : T)$.

In Figure 8.11 we give an example that shows the stages of the EVRSP. Figure 8.11.a shows the Cartesian product of components G_1, G_3 given in Figure 8.9. The dotted arcs in Figure 8.11.b are selected for removal. For the arcs u_1u_3 and u_3u_5 both with label $c¿_1x:T$, there exists a related arc u_1u_2 with label $c¡x:T$. Then, because $P_1 = u_1u_2, P_2 = u_1 \cdots u_6$, $P_1(c¿_1x:T) = 1 > P_1(c¡x:T) = 0$ and $P_1(c¿_1x:T) = 1 \leqslant P_2(c¡x:T) = 2$, u_1u_3 and u_3u_5 are removed in Figure 8.11.c. The last stage of the EVRSP removes u_3, u_5 and the arcs that have u_3, u_5 as a tail because $d^-_{G_1\square G_3}(u_3) = d^-_{G_1\square G_3}(u_5) = 1$ and $d^-_{G_1\overset{\diamond}{\boxtimes}G_3}(u_3) = d^-_{G_1\overset{\diamond}{\boxtimes}G_3}(u_5) = 0$, which leads to Figure 8.11.d.

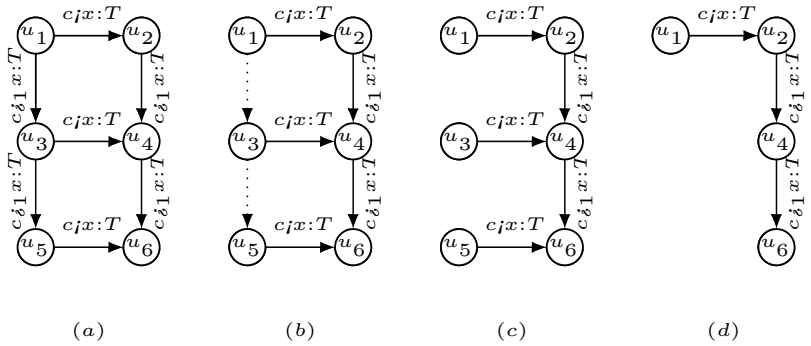


Figure 8.11: The EVRSP from $G_1\square G_3$ (a), two stages of $G_1\overset{\diamond}{\boxtimes}G_3$ (b, c), to $G_1\overset{\diamond}{\boxtimes}G_3$ (d).

8.2.3 Case Study of the Extended Half-Synchronous Alphabetised Parallel Operator

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

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

Assume that the total data used by these threads does not fit in the L2 cache, therefore every context switch leads to a cache flush. This increases the context-switch time (Li et al., 2007). According to Li et al. (2007) due to L2 cache flushes the context-switch time can take up to 1.5 ms for the hardware and software under consideration. In average Li et al. (2007) measured a context-switch time of 3.8 μs.

Taking into account the measured timing for a context switch, we assume that the

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

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

To calculate the values that drive the actuators, the Controller threads interact with an Algorithmic Software process ($Alg.Soft.$). The Algorithmic Software process calculates, for example, the FFT of the data by communicating via memory mapped I/O to an FPGA. The FPGA performs a FFT on the data. This architecture is shown in Figure 8.12.

Furthermore, assume that

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

This leads to a simple CSP specification given in Listing 8.8 using the extended half-synchronous operator, the ¡-actions and the indexed ¿$_i$-actions, where the alphabet of CES is CES, the alphabet of A_i is A_i and the alphabet of C_j is C_j.

Remark 8.2.5. The c_2¡*stop* of A_1 and A_2 are asynchronous writes. Because both A_1 and A_2 perform this action and the C_1 and C_2 read this action only once, one of the writes is not read. This is an example of a writing without reading, which is intended, as the C_1 and C_2 have to start *stopping* as soon as possible.

Remark 8.2.6. Because the reads have different indexes, the C_1 and C_2 do not delay one another.

Remark 8.2.7. The c_1-channel is unidirectional because CES only writes to A_1 and A_2. The c_2-channel is bidirectional because A_1 and A_2 write to C_1 and C_2 and vice versa.

The graphs representing the processes in Listing 8.8 are given in Figure 8.13. The behaviour not modelled in Listing 8.8, the $\cdots$, are left out of Figure 8.13.

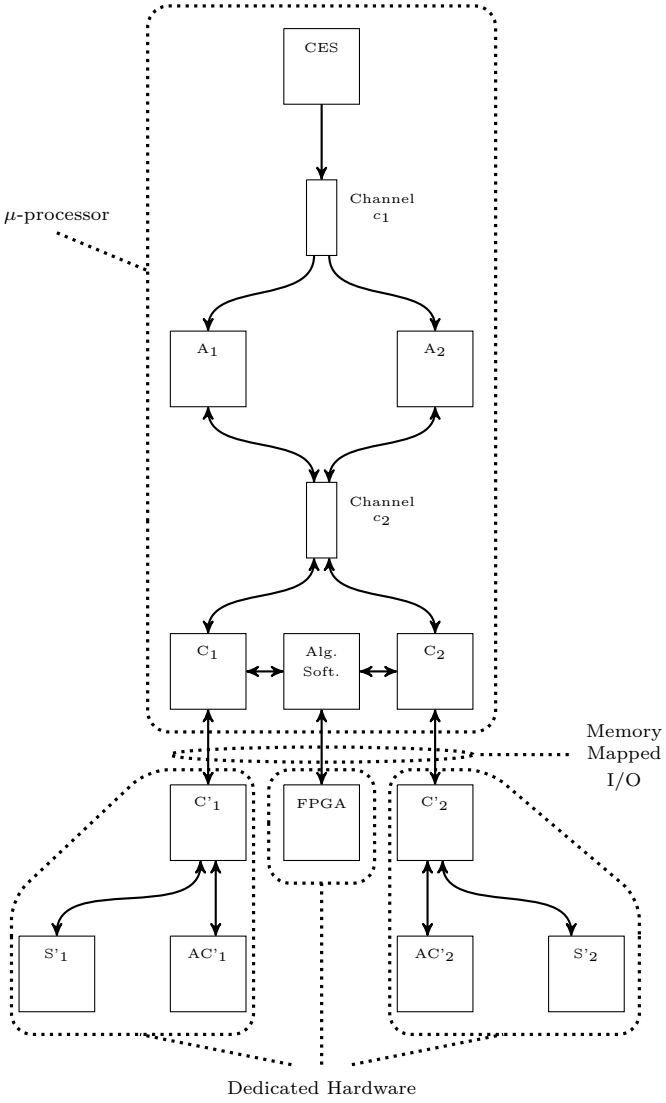


Figure 8.12: Communication Flow of the Controlled Emergency Stop.

Remark 8.2.8. It is up to the process software to handle the state transitions. This includes the handling of guarded actions, which are labels in the graph.

The processes C_1, C_2 in Listing 8.8 are synchronising over the $c_2¿_1$*boot*-action and *waitForNextPeriod*-action. Only the *waitForNextPeriod*-action occurs in all longest paths. But still the worst-case performance is improved by the execution time of one *waitForNextPeriod*-action, together with two context switches.

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

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

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

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

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

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

Listing 8.8: The Controlled Emergency Stop Process Specification.

Therefore for the EVRSP of C_1, C_2 and $C_1 \overset{\diamond}{\boxtimes} C_2$, there is some gain. The memory occupancy is not quadratic with respect to the number of vertices of C_1 and C_2, because of the order that the ¡-actions and ¿-actions impose on the product. For A_1, A_2 and CES the gain is better, because both the *ack*-action and *waitForNext-Period*-action are on all longest paths.

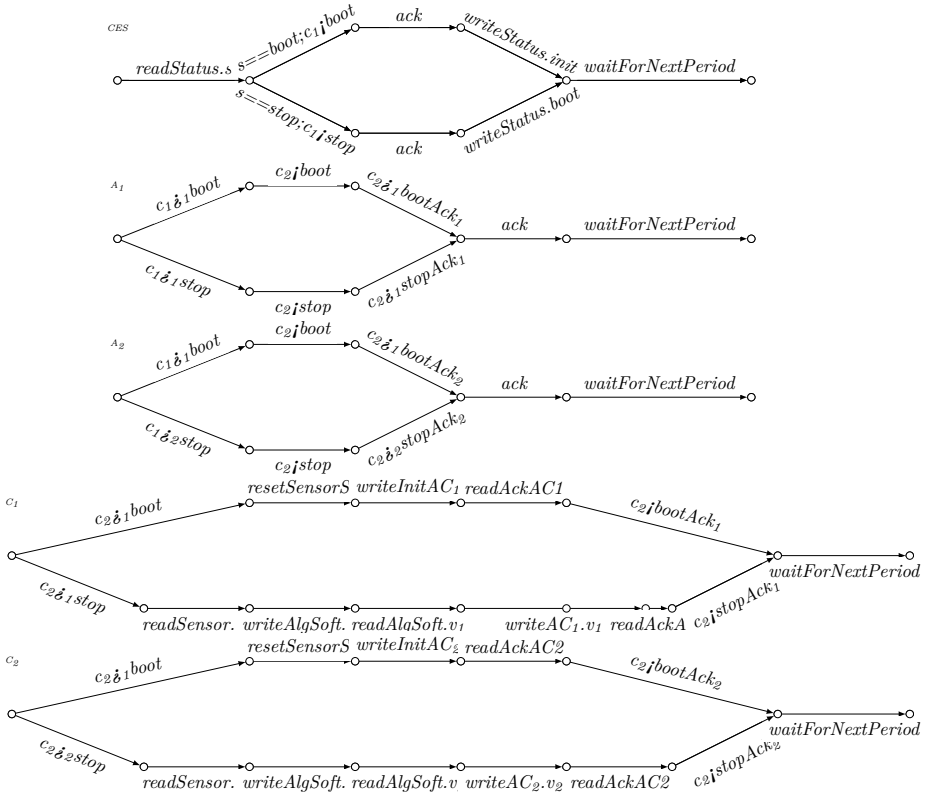


Figure 8.13: Graphs CES, A_1,A_2, C_1 and C_2.

For example, in Figure 8.14 a longest path of $A_1 \overset{\diamond}{\boxtimes} A_2$ contains eight arcs, whereas a longest path of A_1 plus a longest path of A_2 is equal to 10. This reduces the overhead of synchronisation considerably. Also, the memory occupancy with respect to the number of vertices and arcs is 26 vertices and 39 arcs for $A_1 \overset{\diamond}{\boxtimes} A_2$ and 16 vertices and 16 arcs (two times 8 vertices and 8 arcs) for A_1 and A_2.

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

One trace given in Listing 8.9, is of particular interest because it shows a longest

path in the combined graph representing the *System* process for a *stop*-action and a *boot*-action shown in Listing 8.8.

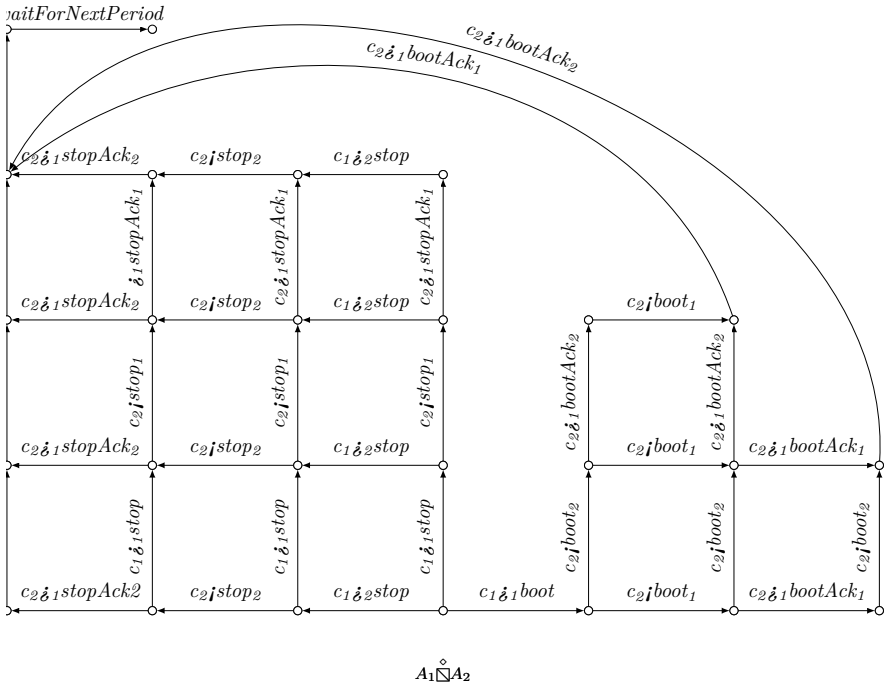


Figure 8.14: Graph $A_1 \overset{\diamond}{\boxtimes} A_2$.

$readStatus.s, 20 \rightarrow c_1!stop, 20 \rightarrow c_1?_1 stop, 20 \rightarrow c_1?_2 stop, 20 \rightarrow c_2!stop, 20 \rightarrow c_2?_1 stop,$
$20 \rightarrow readSensor.s_1, 80 \rightarrow c_2?_2 stop, 20 \rightarrow readSensor.s_2, 80 \rightarrow c_2!stop, 20 \rightarrow writeAlg$
$Soft.s_1, 130 \rightarrow writeAlgSoft.s_2, 130 \rightarrow readAlgSoft.v_1, 20 \rightarrow writeAC_1.v_1, 160 \rightarrow read$
$AckAC1, 20 \rightarrow c_2!stopAck_1, 20 \rightarrow c_2?_1 stopAck_1, 20 \rightarrow readAlgSoft.v_2, 20 \rightarrow writeAC_2$
$.v_2, 160 \rightarrow readAckAC2, 20 \rightarrow c_2!stopAck_2, 20 \rightarrow c_2?_1 stopAck_2, 20 \rightarrow ack, 60$[12]
$\rightarrow writeStatus.boot, 20 \rightarrow waitForNextPeriod, 100$ [13]$\rightarrow SKIP$

Listing 8.9: Trace of the CES.

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

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

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

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

Although there is no deadline-miss in this fictive example for the stop part of the CES, when the model would support the writing to and reading from buffers, the best-case execution time increases. For example, when adding three buffers where each buffer has two actions to perform, there is an extra 120 μs execution time. This leads to an execution time in the best case of 1060 μs. Then a deadline-miss seems inevitable.

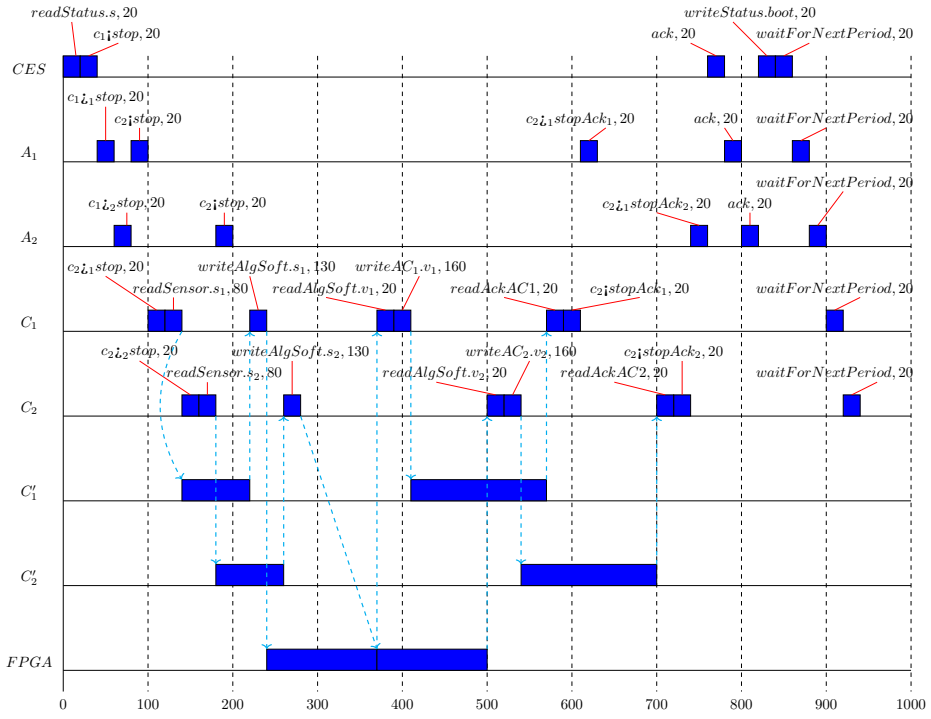


Figure 8.15: Time line of the *stop*-part of the Controlled Emergency Stop.

In Figure 8.15 the time line of a possible trace of the *stop* part of the CES is given. Each grey block represents the time that the thread is executing. The label of each hardware related action contains the overall time. If applicable, this includes the time the hardware needs to reply. The dashed arrows represent a call to the hardware and the reply from the hardware.

The stop part of the CES takes 940 μs to execute (Figure 8.15). This can be improved by using the EVRSP of the graphs, $SynchronisedSystem = CES \diamond\!\!\!\!\!\!\diagup A_1 \diamond\!\!\!\!\!\!\diagup A_2 \diamond\!\!\!\!\!\!\diagup C_1 \diamond\!\!\!\!\!\!\diagup C_2$. The actions that synchronise are $waitForNextPeriod$

and *ack*, therefore the processor needs at most 820 μs to execute the thread represented by the graph *SynchronisedSystem*.

The improvement with respect to timeliness can be easily seen when we model the CES using standard CSP as shown in Figure 8.16, although this example gives an improvement of only 50μs. Obviously, in a more complex standard-CSP example where buffers have to be modelled, the performance would be significantly worse. The $c_2.stop, 20$ actions of A_1, A_2, C_1 and C_2 are executed atomically, therefore it is immaterial which of the processes A_1, A_2, C_1 and C_2 executes the action $c_2.stop, 20$ first. In fact the priority inheritance protocol (Sha et al., 1990) is implemented for the processes A_1, A_2 and C_1, C_2 for the action $c_2.stop, 20$.

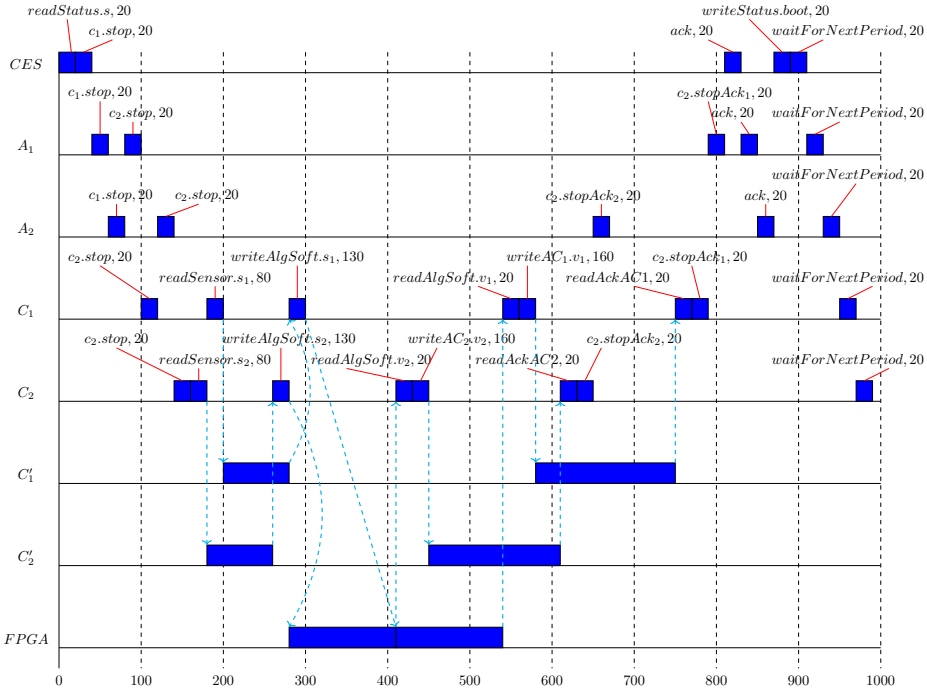


Figure 8.16: Time line of the *stop*-part of the Controlled Emergency Stop without asynchronous readers and writers.

8.3 Discussion and Conclusions

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

process does not have to wait for the reading process to synchronise. There are three advantages of the $_X\Downarrow_Y$ operator in combination with the DVRSP

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

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

Secondly, we have discussed an extension of the $_X\Downarrow_Y$ operator, the new $_X\Updownarrow_Y$ operator and the ¡-*action* together with the new ¿$_i$-*action*, that delay the reading of a process from a buffer. The $_X\Updownarrow_Y$ operator together with the ¡-*action* and ¿$_i$-*action* are an abstraction of a buffer, therefore the designer does not have to model the buffer as well. In this manner, the writing process does not have to wait for the reading process to synchronise. There are five advantages of the $_X\Updownarrow_Y$ operator in combination with the EVRSP with respect to standard CSP:

1. it eases the design by taking away the burden of separating the writing actions and reading actions in time, which eliminates the necessity of a buffer,
2. it gives maximum flexibility by indexing the reading actions,
3. it allows multiple writing actions to the same channel,
4. the length of the longest paths is reduced, if the writing actions and reading actions are part of all the longest paths of the participating graphs,
5. in a distributed computing system, for example, a processor-coprocessor combination, the waiting time of the processor or coprocessor can be reduced.

The first advantage makes the design less error-prone and therefore the design phase needs less time. The absence of a buffer leads to fewer actions that have to be performed by the involved threads and therefore to a reduction of the utilisation of the processor,

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

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

The fifth advantage is due to a reduction of the end-to-end execution time during

one period and therefore leads to an application for which the possibility of a deadline-miss is reduced.

If no buffer is needed in the design, the utilisation of the processor remains the same with respect to using synchronous writing actions and reading actions, because the disconnection of reading actions and writing actions still leads to threads that execute these actions. In such a situation, the disconnection of reading actions and writing actions is especially useful if there is no gain possible by using the EVRSP. This happens when the multiplication of any two components leads to a memory occupancy that exceeds the available memory in the target system.

9

Conclusions and Recommendations

In this thesis, we have shown that a considerable performance gain can be achieved for Periodic Hard Real-Time Control Systems (PHRCSs) developed with process algebras using graph theory. The performance gain originates in two aspects of PHRCSs: reduction of the overhead by context switches and reduction of the end-to-end processing time of Periodic Hard Real-Time Control Processes (PHRCPs). To reach this goal we have introduced several forms of a synchronised product: the Vertex-Removing Synchronised Product (VRSP), the Dot Vertex-Removing Synchronised Product (DVRSP) and the Extended Dot Vertex-Removing Synchronised Product (EVRSP). Furthermore, we have studied the graph-theoretical and number-theoretic aspects of these synchronised products. This has led to three research questions of which the conclusions and evaluation are elaborated in Section 9.1 through Section 9.3. We finish with a preview of future work in Section 9.4.

9.1 Reduction of Context Switches

To answer the first research question,"How can the *number of context switches* be reduced for periodic hard real-time systems, which are developed using process algebras, by means of a graph-theoretical approach in such a manner that the *performance* of the system is improved?", we introduced a Vertex-Removing Synchronised Product (VRSP) for which we proved in Theorem 3.6.4 that the length of two components G_1 and G_2 can be reduced if all longest paths of one component, say G_1 and at least one longest path of the other component, say G_2, contain at least one synchronising arc. Together with transformation functions T, T^{-1}, which map the processes onto graphs and vice versa, this shows that we can reduce the worst-case performance of periodic real-time parallel processes, by combining two processes, where all longest traces for one process must contain synchronising actions and the other process must contain at least one longest trace with at least one synchronising action.

To prove Theorem 3.6.4 we have introduced a VRSP for which we created four stages to ease the prove of Theorem 3.6.4:

- The first stage is the Cartesian product. The source and sink of the Cartesian

product are used to identify which vertices and their arcs have to be removed in the third and fourth stage.

- The second stage is the weak synchronised product. For the weak synchronised product, we were able to identify a pathological case in a natural manner. This made it visible that a set of parallel processes may contain unwanted behaviour, e.g. a deadlocked state. We have shown in the proof of Lemma 3.6.1 and Remark 3.4.1, that we can filter out this unwanted or ill-defined behaviour of processes by finding paths that shrink under the weak synchronised product.
- The third stage, the reduced weak synchronised product removes the vertices and their arcs for which the in-degree is zero, whereas the in-degree of the vertex in the Cartesian product has an in-degree greater than zero.
- The fourth stage replaces quadrangles with identical labels by their diagonal arc, after which again vertices and their arcs for which the in-degree is zero, whereas the in-degree of the vertex in the Cartesian product has an in-degree greater than zero, are removed.

We informally introduced the notion of a consistent and an inconsistent set of graphs (representing real-time periodic processes) in Chapter 3 and formalised this notion in Chapter 6. Consistency is based on the contraction of components together with the sink and the source, where the sink and the source have to be invariant over the graph multiplication by the VRSP.

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. But still, if the Periodic Hard Real-Time Control System (PHRCS) does not fulfil the requirements with respect to the deadline of its Periodic Hard Real-Time Control Processes (PHRCPs), calculating all possible products of two or more components may produce a set of components for which the processes they represent comprise a PHRCS that *will* fulfil the requirements with respect to deadline and memory occupancy.

To increase the chance that such a PHRCS exist, we decompose the components. Decomposition of the components gives a set of components from which the VRSP can be taken. If all combinations of components are calculated, this will lead to sets of components of which a subset will contain solutions that fulfil the requirements with respect to the deadline and the memory occupancy of the PHRCPs. From this set, the designer has to choose a solution that is most feasible with respect to the other requirements or future design. For example, if it is foreseen that new non-hard real-time functionality is required, the designer may choose the solution consuming the least memory as this new functionality will not be a burden to the deadline of the PHRCS.

In general, 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

has to be taken for the new process specification. Furthermore, the number of parallel processes, and therefore the number of components of the graph G, is often limited to 15 or 20 processes. For 15 processes, there are $B_{15} \approx 10^9$ vertices in the related lattice, where each vertex in the lattice represents a unique set of components representing the PHRCPs of the PHRCS. But for 20 processes there are $B_{20} \approx 5 \cdot 10^{13}$ vertices in the lattice. Depending on the speed of the computing system it may take several days to calculate an optimal solution out of all partitions for 20 processes (assuming the algorithm that calculates an optimal solution uses not more than the available memory to store the intermediate data). For each extra process, this 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. For larger sets of processes, we have developed heuristics that will calculate a set of components by continuously multiplying two components by the VRSP until a graph is obtained comprising only one component. The drawback of these heuristics is that they may miss a solution and only calculate sets of components that have a deadline miss or do not fit in the available memory.

Furthermore, we have achieved the goal that a set of processes that does not meet its deadline or does not fit in the available memory can be transformed by decomposition and graph multiplication by the VRSP into a set of processes that will fulfil both requirements, or if the set of processes does not fulfil both requirements then the designer has to redesign the set of processes or more powerful hardware has to be used.

Clearly, for applications containing hundreds of processes heuristics must be used that will give an educated guess which partitions have 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. FDR3, this should not be an issue.

We finished the first research question, “How can the *number of context switches* be reduced for periodic hard real-time systems, which are developed using process algebras, by means of a graph-theoretical approach in such a manner that the *performance* of the system is improved?”, with a small case study, the Production Cell, in the reproduction of our paper (Boode and Broenink, 2014) in Chapter 4 which showed the advantages of our graph product.

9.2 Graph-Theoretical Properties of the Reduction Operator

The second research question, "What are the *algebraic and graph-theoretical properties of the graph-theoretical approach*, which are developed using process algebras, that reduce the number of context-switches for periodic hard real-time systems in such a manner that the performance of the system is improved?", is mainly based on graph theory and is meant to give an inventory of these graph-theoretical properties. We have given proof and examples that show the correctness and usability of the reduction operator VRSP.

The proofs strongly rely on the notion of consistency. When the set of components and their products are pairwise consistent then the set of components is commutative, idempotent and associative. Furthermore, we have shown that the VRSP does not distribute over the + operator.

Consistency is essential because otherwise, the VRSP would not be associative. Apart from e.g. deadlocks for a set of processes, the number of solutions for a set of components representing these processes follows the Bessel number ($\tilde{B}_n$) series, as we have proved in Chapter 5. The numbers in the Bessel number series are a magnitude larger than the numbers in the Bell number (B_n) series and this is another reason why associativity is necessary. Finding a solution for a non-associative set of components under the VRSP is only feasible for a much smaller set of components with respect to an associative set of components under the VRSP. For example, when associativity applies for a set of 16 components a solution can be found in a reasonable amount of time, whereas for a non-associative set of components this would be 11 components only (Table 5.1 on page 64).

Another improvement that we achieved is the development of theory to factor the components in sub-components and use the VRSP on these components. We constructed and proved theorems that decompose components. Factorisation of a set of pairwise consistent components and their products, into their factors, is again a set of components for which this set of components and their products are pairwise consistent under the VRSP. Although this enlarges the number of combinations of components, this may give a solution that is not available in the original set of components.

9.3 End-to-end Processing-Time Reduction Operator

The third research question, "How can the reduction of the *end-to-end processing time* of a set of processes during every period of any periodic hard real-time system, which is developed using the *process algebra Communicating Sequential Processes (CSP)*, be achieved by means of an *extension* of the graph-theoretical approach of research question one?", has led to two proposals for a new operator and the related writing action and reading action on process-algebraic level. Based on these proposals, two graph products are given for which a considerable performance improvement can be achieved.

The end-to-end processing time reduction deals with two aspects of passing information between processes, synchronous and asynchronous. In case of asynchronous communication, a process is allowed to pass information and continue execution, whereas synchronous communication follows the rendezvous design pattern which leads to a suspension of the sending process till the receiving process has finished handling the passed information by sending an acknowledgement.

In Chapter 8 we have discussed a new $_X\Downarrow_Y$ operator together with the new ¡-*action* and ¿-*action*, which delay the reading of a process from a buffer. We have extended the $_X\Downarrow_Y$ operator and the ¡-*action* and ¿-*action* with the introduction of the $_X\Updownarrow_Y$ operator and the ¡-*action* together with the new ¿$_i$-*action*. The $_X\Updownarrow_Y$ operator together with the ¡-*action* and ¿$_i$-*action* are an abstraction of a buffer, therefore the designer does not have to model the buffer as well. In this manner, the writing process does not have to wait for the reading process to synchronise.

The advantages of the $_X\Updownarrow_Y$ operator in combination with the Extended Dot Vertex-Removing Synchronised Product (EVRSP) with respect to standard CSP are that it eliminates the necessity of a buffer, while at the same time it allows multiple writing actions to the same channel, reduces the length of the longest path and reduces the end-to-end processing time of processes. Furthermore, the design is less error-prone and therefore the design, implementation and test phase need less time.

If no buffer is needed in the design, the utilisation of the processor remains the same with respect to using synchronous writing actions and synchronous reading actions, because the disconnection of reading actions and writing actions still leads to threads that execute these actions. In such a situation, the disconnection of reading actions and writing actions is especially useful if there is no more gain possible by using the VRSP. This happens when the multiplication of any two components leads to a memory occupancy that exceeds the available memory in the target system.

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

9.4 Future Work

Graph-theoretical point of view

This research is restricted to periodic hard real-time applications, where the periods, release times and deadlines are the same. This can be extended to applications where this is *not* the case. Such a relaxation of the set of PHRCPs leads to a Real-Time System (RTS) where, for example, if the RTS is an event driven system, the real-time requirements are met because only the events and not the tasks have a hard deadline.

For the VRSP and the EVRSP this relaxation can only be achieved by relaxing

the constraints on the components we are considering. Because we are only taking into account finite, deterministic, labelled, acyclic, directed multigraphs, we can remove the constraint that our graphs have to be acyclic. But then we have to reconsider the definition of the VRSP and the EVRSP because the VRSP and the EVRSP are based on a sink and a source of the Cartesian product of a pair of components, which relies on the components being acyclic. For the same reason, the definition of consistency of components has to be adapted. Furthermore, to construct a model of an event driven RTS, we have to adapt the labels of the arcs of the components in such a manner that besides the execution time of the event also the relative deadline of the event is taken into account. But then the VRSP and therefore the EVRSP have to be redefined, because the choice out of two asynchronous arcs (belonging to different components) representing these events, is not free any more, but depends on the laxity[1] of the event.

Another issue is that the periods of all processes are the same. Lifting this requirement to any period for all processes, leads to scheduling issues, because we cannot adapt the VRSP and the EVRSP in such a manner that, for instance, by taking the Least Common Multiple of the periods, we create the VRSP and the EVRSP of sequences of components.

A totally different area would be the extension into a system containing one or more multi-core processors. Because for such systems the synchronisation of processes, distributed over several cores and possibly in different processors, by the synchronisation software will consume more time. Therefore the processes have to be distributed over the processors and their cores in such a manner that synchronisation of actions of processes in different processors and in different cores on the longest paths of the processes is minimised, while maximising the synchronisation of processes in the same core. Ultimately, even processes can be distributed over different cores, possibly on different processors. This will lead to a major change of the VRSP and the EVRSP and adjustment of the heuristics calculating the VRSP and the EVRSP. For such a change, the theory we have developed must be expanded, so that the characteristics with respect to consistency of graphs and the decomposability of graphs are meaningful and the related theorems are proved.

All these issues are related to Cyber-Physical Systems (CPSs). But there are also open issues for the two decomposition theorems we developed. Although we can decompose a graph using these two theorems, the decompositions are in general not decompositions into prime graphs. For example, the two theorems do not decompose a graph which contains a subgraph that can be decomposed into subgraphs for which their Cartesian product is isomorphic to the original subgraph. Therefore, theory has to be developed similar to the seminal paper of Sabidussi (1960) on the decomposition of graphs by the Cartesian product and we have to prove that this decomposition theorem together with our two decomposition

[1] The laxity of an event is the relative deadline of the event minus the remaining execution time of the event.

theorems are the only decompositions possible and lead to a decomposition into prime graphs.

Application point of view

Although we have dealt in this thesis with the most important issues, several issues in our design cycle have not been addressed yet.

With respect to system architectures, we have mentioned in Boode and Broenink (2014) the necessity of the transformation functions T and T^{-1} (Figure 4.2 on page 45), which transform a process algebraic specification into a set of graphs and vice versa. However, these transformation functions are not defined yet. Therefore an interface to any tool-chain is missing.

Perhaps the most important issue in the lack of tooling is the development of an interface to the TERRA tool; an integration of the VRSP, or ultimately the EVRSP, into the TERRA tool seems necessary. For the TERRA tool the model of computation is based on the *vertex* of a component as being the point where the required behaviour is calculated, whereas for the VRSP and the EVRSP the behaviour is represented by the *arcs* of the component. These two views are the dual of one-another and require either a change of the VRSP and the EVRSP, or a converter converting a set of the VRSP (EVRSP) components into TERRA models and vice versa. Because TERRA can produce machine readable CSP, CSP_M (Scattergood, 1998), it is obvious that the transformation function T can be straightforward, but the transformation function T^{-1} has to map a graph representing a CSP specification onto a TERRA model, which contains structures not available in CSP. Therefore, it is not obvious that such a transformation function is possible without a change of the TERRA tool.

Furthermore, there is no fully operational tool-chain that automatically, based on the process-algebraic specification together with the VRSP, produces software which can be compiled and built, thereby producing a set of PHRCPs. Also, tooling that supports the choice for synchronous writing actions and reading actions versus the EVRSP has to be developed, this all, in spite of the implementation by de Boer (2016) of the VRSP.

The end result to go for could be allowing cyclic and non-deterministic process specifications distributed over several multi-core processors and study the impact on the VRSP and the EVRSP, leading to an implementation into the TERRA tool.

Acknowledgements

First of all, I want to thank my beloved wife Gjoera for her patience and support during the research of the last six years. Without that, it would not have been possible to finish this thesis.

Naturally, I want to thank Peter, Herman and Cock, whose letters of recommendation convinced the executive board of the InHolland University of Applied Science that my research proposal was of value to our school and that time to perform the research should be awarded.

Therefore, I am in great debt to the Department of Computer Science, in particular to Dirk and Egbert, who made this all possible by allowing me to work on my thesis for two to three days a week during the research period.

Next, I want to thank Stefano for giving me the opportunity to perform my research at the Robotics and Mechatronics Group.

Of course, the support of Jan with respect to real-time control systems and Hajo with respect to graph theory was essential for this research; without it, this thesis would have lacked the quality necessary to obtain a doctorate. Jan and Hajo, thank you very much!

My gratitude also goes to the members of the promotion committee, Brian, Cock, Gerard, Herman and Nelly, for reading and accepting this thesis. Reading a thesis takes a considerable amount of time of which most scientists never have enough, thank you all, it is most appreciated.

Also, I want to thank Jolanda for her support on all small things like booking flights, hotels, printing papers, and most of all for borrowing the heater when the temperature in our rooms were below 20° C.

Last, but certainly not least, I want to thank my para-nymphs Raoul and Jaap for their moral support before and during the defence of my thesis.

Appendices

I Choice in Two Parallel Processes

In Listing 1, we give an example of the serialisation of two processes containing choice. Two processes synchronise over the actions a, c, and e. According the process specification of Listing 1, two traces can occur, $d \rightarrow c \rightarrow e$ and $a \rightarrow b \rightarrow e$.

$$
\begin{array}{lcl}
G_1 & = & (a \rightarrow b \rightarrow G_1') \\
 & & \Box \\
 & & (d \rightarrow c \rightarrow G_1') \\
G_1' & = & e \rightarrow Skip; \\
 & & \\
G_2 & = & (a \rightarrow G_2') \\
 & & \Box \\
 & & (c \rightarrow G_2') \\
G_2' & = & e \rightarrow Skip; \\
G & = & G_1 \ {}_{\{a,b,c,d,e\}} \parallel_{\{a,c,e\}} \ G_2
\end{array}
$$

Listing 1: Description of the *CHOICE* in two parallel processes.

The last stage of Figure 1 shows the graph representing these two traces.

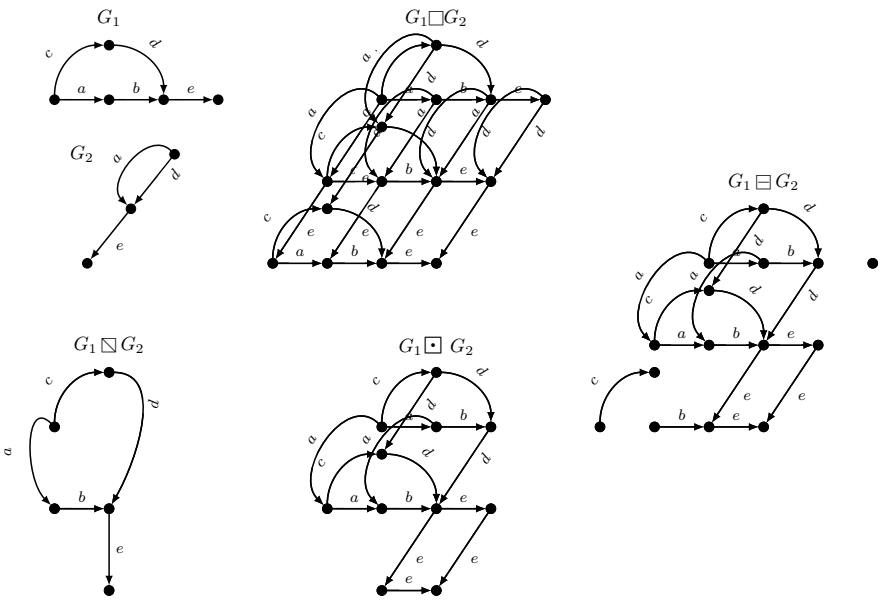


Figure 1: Choice in two parallel processes, from $+$ via $\Box, \boxminus$, $\boxdot$ to $\boxtimes$.

Assuming that the weight of all arcs is 1, the graph consisting of the two components G_1 and G_2 has 8 vertices and a length $\ell(G_1)+\ell(G_2)=5$, whereas the synchronised product $\ell(G_1 \boxtimes G_2)$ has 5 vertices and a length $\ell(G_1 \boxtimes G_2)=3$. This example shows a gain for both the memory occupancy, as the performance of the application.

II Complexity of the Number of Full Paths

Let a graph G with multiplicity $m_{i,j} \leqslant m$ for arcs $v_i v_j$ consist of vertices $v_1, \ldots, v_{n+1}$ with $d^-(v_1)=0, d^-(v_{i+1})>0, d^+(v_i)>0, d^+(v_{n+1})=0$ and arcs $v_i v_j, 0<i<j \leqslant n+1$. Then the number of full paths (from the source to the sink) in G are, for $V(G)=\{v_1,v_2\}$, not more than m full paths from v_1 to v_2, for $V(G)=\{v_1,v_2,v_3\}$ not more than $m+m^2$ full paths from v_1 to v_3 (i.e. $v_1v_2v_3, v_1v_3$), such that the number of full paths that end in v_{n+1} is not more than m times the summation over the number of paths starting in v_1 and ending in v_1 through v_n plus one. Hence let f_i be the number of paths starting in v_1 and ending in v_i then $f_n \leqslant m(1+f_1+f_2+\ldots+f_{n-1})$, with $f_1=0, f_2=m_{1,2} \leqslant m, n \geqslant 2$. Then $f_n+f_{n-1} \leqslant m(1+f_1+\ldots+f_{n-1})+m(1+f_1+\ldots+f_{n-2}) = 2m(1+f_1+\ldots+f_{n-2})+mf_{n-1} = 2f_{n-1}+mf_{n-1} \Rightarrow f_n \leqslant (m+1)f_{n-1} \Rightarrow f_n \leqslant (m+1)^{n-2}.f_2 \leqslant m(m+1)^{n-2}$.

III Algorithms

In Chapter 4, we have given an overview of a software architecture in which our VRSP reduces the number of context switches (Figure 4.1 on page 44).

In Chapter 6 we have defined the Vertex-Removing Synchronised Product (VRSP).

A classification for the associativity of graph products is given by Imrich and Izbicki (1975). Based on this classification, Hammack et al. (2011) give an incidence function for the multiplication table for associative graph products, $\delta : V(G) \times V(G) \to \{\Delta, 1, 0\}$, with

$$\delta(g,g') = \begin{cases} \Delta & \text{if } g = g', \\ 1 & \text{if } g \neq g' \text{ and } gg' \in E(G), \\ 0 & \text{if } g \neq g' \text{ and } gg' \notin E(G). \end{cases}$$

As Hammack et al. (2011) observes, there should be some definite rule that determines the edge set of the product. In our case we can establish such a function for the arc set of the intermediate stage of the VRSP, but not for the VRSP, as we will show in the sequel.

The heuristics in the appendix of Boode and Broenink (2014) use functions $\delta(g,g') : V(G) \times V(G) \to \{\Delta, 0, 1\}$ and $\delta_{int}(g,g') : V(G) \times V(G) \to \{\Delta, 0, 1^a, 1^s\}$ that calculate the Cartesian product and the intermediate stage of the VRSP.

For the intermediate stage of the VRSP we have an adapted version of the incidence function of Hammack et al. (2011). The (a)synchronous arc set of a graph is always relative to the (a)synchronous arc set of another graph. Therefore our incidence function for a graph G_1 is always relative to a graph G_2. For the intermediate

stage of the VRSP the incidence function is defined as $\delta_{int} : V(G_1) \times_{G_2} V(G_1) \to \{\Delta, 0, 1^a, 1^s\}$, with

$$\delta_{int}(g, g') = \begin{cases} \Delta & \text{if } g = g', \\ 0 & \text{if } g \neq g' \text{ and } gg' \notin A(G_1), \\ 1^a & \text{if } g \neq g' \text{ and } gg' \in A^a(G_1) \ (\lambda(gg') \notin L(G_2)), \\ 1^s & \text{if } g \neq g' \text{ and } gg' \in A^s(G_1) \ (\lambda(gg') \in L(G_2)). \end{cases}$$

We give in table 1 the multiplication table for the intermediate stage of the VRSP $\boxtimes$ using the incidence function δ_{int}. In Table 1 the Δ represents a single vertex ($g = g'$), the 0 represents two not adjacent vertices and the $1, 1^a$ or 1^s represent two adjacent vertices.

$G_1 \boxtimes G_2$	Δ	0	1^a	1^s
Δ	Δ	0	1	0
0	0	0	0	0
1^a	1	0	0	0
1^s	0	0	0	1

Table 1: The intermediate stage of the VRSP.

As observed by Hammack et al. (2011), if the second row and second column contain only zeros, then the product is associative.

For the VRSP we cannot define the δ_{syn} function because for $g = g'$, $\delta_{syn}(g, g') = \Delta$ or $\delta_{syn}(g, g') = \varnothing$ in the VRSP. This depends on whether the vertex g in the intermediate stage of the VRSP is removed or not when calculating the VRSP.

Figure 4.6 on page 48 shows an example of non-associative components under the VRSP. Therefore, the associativity of the VRSP is not trivial. The incidence function δ_{int} is used in the algorithms that calculate the intermediate stage of the VRSP of two components G_i and G_j.

In Algorithm 4, we describe the general structure of how to implement the algorithm, which contains a call to the specific calculation method *calcAlgorithm*(G). In Algorithm 4 the subroutine *pairwiseConsistent*(G) checks for a graph $G = \sum_i G_i$ 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 out-flowing arcs. The subroutines *calcCartSize* and *calcSyncProd* are (worst case) the product of the vertex and arc sizes.

Algorithm 1 calculates the Cartesian product, Algorithm 2 calculates the intermediate product and Algorithm 3 calculates the synchronised product of two components G_i and G_j.

The pseudo-code of the Largest Alphabetical Intersection (LAI) is given in Al-

gorithm 5. Because the pseudo-code of the other two *calcAlgorithm*(G)'s is likewise straightforward, they are left out.

The pseudo-code of the Minimal Memory Occupancy (MMO) is given in Algorithm 6. MMO is almost identical to LAI, but requires more computation, as the VRSP of all products has to be calculated. No attempt has been made to optimise these algorithms, although that is necessary for usage in a tool-chain.

Algorithm 1 Calculating the Cartesian product

Require: G_i, G_j
1: $V(G_i \square G_j) = V(G_i) \times V(G_j)$
2: $A(G_i \square G_j) = \varnothing$
3: **for each** $g_i, g_i' \in V(G_i)$ **and** $g \in V(G_j)$ **do**
4: **switch** $(\delta(g_i, g_i'))$
5: **case** Δ, 0:
6: **break**
7: **case** 1:
8: $A(G_i \square G_j) = A(G_i \square G_j) \bigcup \{(g_i, g)(g_i', g)\}$
9: **for each** $\lambda(g_i g_i') \in L(G_i)$ **do**
10: $L(G_i \square G_j) = L(G_i \square G_j) \bigcup \{\lambda((g_i, g)(g_i', g)) | \lambda((g_i, g)(g_i', g)) = \lambda(g_i g_i')\}$
11: **end switch**
12: **for each** $g_j, g_j' \in V(G_j)$ **and** $g \in V(G_i)$ **do**
13: **switch** $(\delta(g_j, g_j'))$
14: **case** Δ, 0:
15: **break**
16: **case** 1:
17: $A(G_i \square G_j) = A(G_i \square G_j) \bigcup \{(g, g_j)(g, g_j')\}$
18: **for each** $\lambda(g_j g_j') \in L(G_j)$ **do**
19: $L(G_i \square G_j) = L(G_i \square G_j) \bigcup \{\lambda((g, g_j)(g, g_j')) | \lambda((g, g_j)(g, g_j')) = \lambda(g_j g_j')\}$
20: **break**
21: **end switch**

Algorithm 2 Calculating the Intermediate Product

Require: G_i, G_j
$V(G_i \boxtimes G_j) = V(G_i) \times V(G_j)$
$A(G_i \boxtimes G_j) = \varnothing$
for each $g_i, g_i' \in V(G_i)$ **and** $g_j \in V(G_j)$ **do**
  switch $(\delta_{int}(g_i, g_i'))$
  case $\Delta, 0, 1^s$**:**
    break
  case 1^a**:**
    $A(G_i \boxtimes G_j) = A(G_i \boxtimes G_j) \bigcup \{(g_i, g_j)(g_i', g_j)\}$
    for each $\lambda(g_i g_i') \in L(G_i)$ **do**
      $L(G_i \boxtimes G_j) = L(G_i \boxtimes G_j) \bigcup \{\lambda(g_i, g_j)(g_i', g_j) | \lambda(g_i, g_j)(g_i', g_j) = \lambda(g_i g_i')$
    break
  end switch
for each $g_j, g_j' \in V(G_j)$ **and** $g_i \in V(G_i)$ **do**
  switch $(\delta_{int}(g_j, g_j'))$
  case $\Delta, 0, 1^s$**:**
    break
  case 1^a**:**
    $A(G_i \boxtimes G_j) = A(G_i \boxtimes G_j) \bigcup \{(g_i, g_j)(g_i, g_j')\}$
    for each $\lambda(g_j g_j') \in L(G_j)$ **do**
      $L(G_i \boxtimes G_j) = L(G_i \boxtimes G_j) \bigcup \{\lambda(g_i, g_j)(g_i, g_j') | \lambda(g_i, g_j)(g_i, g_j') = \lambda(g_j g_j')$
    break
  end switch
for each $g_i, g_i' \in V(G_i)$ **and** $g_j, g_j' \in V(G_j)$ **do**
  switch $(\delta_{int}(g_i, g_i'))$
  case $\Delta, 0, 1^a$**:**
    break
  case 1^s**:**
    switch $(\delta_{int}(g_j, g_j'))$
    case $\Delta, 0, 1^a$**:**
      break
    case 1^s**:**
      $A(G_i \boxtimes G_j) = A(G_i \boxtimes G_j) \bigcup \{(g_i, g_j)(g_i', g_j') | \lambda(g_i, g_i') = \lambda(g_j g_j')\}$
      for each $\lambda(g_i g_i') \in L(G_i)$ **do**
        $L(G_i \boxtimes G_j) = L(G_i \boxtimes G_j) \bigcup \{\lambda(g_i, g_j)(g_i', g_j') | \lambda(g_i, g_j)(g_i', g_j') = \lambda(g_i g_i') = \lambda(g_j g_j')\}$
      break
    end switch
    break
  end switch

Algorithm 3 Calculating the VRSP

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

Algorithm 4 Calculating a General VRSP Heuristic

Require: $G = \sum_{i=1}^{n} G_i$, $\mathcal{D} = deadline$, $\mathcal{M} = available\ memory\ in\ target\ system$

```
1:  sizeG    = calcSize(G)
2:  deadlG = calcDeadline(G)
3:  for i = 1 to n − 1 do
4:    if sizeG ⩽ M and deadlG ⩽ D then
5:      return G
6:    else
7:      if ¬pairwiseConsistent(G) then
8:        return ∅
9:      else
10:       (i, j)      = calcAlgorithm(G)
11:       G           = (G ⋃(G_i ⋉ G_j))\(G_i ⋃ G_j)
12:       sizeG    = sizeG − calcSize(G_i ⋃ G_j) + calcSize(G_i ⋉ G_j)
13:       deadlG = deadlG − calcDeadline(G_i ⋃ G_j) + calcDeadline(G_i ⋉ G_j)
```

Algorithm 5 Calculating the Largest Alphabetical Intersection

Require: $G = \sum_{i=1}^{k} G_i$

```
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(G_i) ⋂ L(G_j)|
7:      if (newNum > num then
8:        num    ← newNum
9:        first   ← i
10:       second ← j
11: return (first, second)
```

Algorithm 6 Calculating the Minimal Memory Occupancy

Require: $G = \sum_{i=1}^{k} G_i$

```
1:  first   = 1
2:  second = 2
3:  mem     = ∞
4:  for i = 1 to k − 1 do
5:    for j = i + 1 to k do
6:      newMEM = m(G_i ⋉ G_j)
7:      if (newMEM < mem) then
8:        mem    ← newMem
9:        first   ← i
10:       second ← j
11: return (first, second)
```

IV Memory versus Deadline Table

With every iteration two components are multiplied using the 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, $\ell(G)$, gives a measure for the number of context switches in the representing processes. The function $m(G)$ calculates the number of vertices and arcs that is used by the graph G. It gives a measure what can be expected as far as the memory occupancy is concerned.

Iteration	Algorithms / $\sum_i \bigotimes_j G_{i,j}$	MNSA		LAI		MSA	
n	Node in lattice	$\ell(G)$	$\mathcal{M}$	$\ell(G)$	$\mathcal{M}$	$\ell(G)$	$\mathcal{M}$
0	$V_{0000000000000000}$	62	175	62	175	62	175
1	$V_{1000000010000000}$	60	169	-	-	-	-
	$V_{0000000001000100}$	-	-	59	182	-	-
	$V_{0000000000000101}$	-	-	-	-	60	183
2	$V_{1200000012000000}$	58	163	-	-	-	-
	$V_{1000000012000200}$	-	-	57	176	-	-
	$V_{0000000001000101}$	-	-	-	-	57	218
3	$V_{1230000012000300}$	56	177	-	-	-	-
	$V_{1200000012000200}$	-	-	55	197	-	-
	$V_{0100000001000101}$	-	-	-	-	55	338
4	$V_{1234000012400300}$	54	171	-	-	-	-
	$V_{1203000012300200}$	-	-	53	191	-	-
	$V_{0100000001100101}$	-	-	-	-	53	475
5	$V_{1234500012450300}$	52	165	-	-	-	-
	$V_{1223000012300200}$	-	-	50	300	-	-
	$V_{0101000001100101}$	-	-	-	-	51	546
6	$V_{1234560012456300}$	50	159	-	-	-	-
	$V_{1223400012340200}$	-	-	48	294	-	-
	$V_{0111000001100101}$	-	-	-	-	49	1,618
7	$V_{1234567012456370}$	48	153	-	-	-	-
	$V_{1223450012345200}$	-	-	46	288	-	-
	$V_{1111000001100101}$	-	-	-	-	48	7,855
8	$V_{1234567812456378}$	47	159	-	-	-	-
	$V_{1223456012345260}$	-	-	44	282	-	-
	$V_{1111000011100101}$	-	-	-	-	46	11,925
9	$V_{1223456712345267}$	43	283	-	-	-	-
	$V_{1223456012345263}$	-	-	42	292	-	-
	$V_{1111000011101101}$	-	-	-	-	45	54,133
10	$V_{1223345612334256}$	42	358	-	-	-	-
	$V_{1223345012334253}$	-	-	41	484	-	-
	$V_{1111100011101101}$	-	-	-	-	43	242,771
11	$V_{1222234512223245}$	41	4,381	-	-	-	-
	$V_{1222234012223242}$	-	-	40	11,978	-	-
	$V_{1111110011101101}$	-	-	-	-	42	367,945
12	$V_{1222233412223234}$	39	4,563	-	-	-	-
	$V_{1222234312223242}$	-	-	39	11,990	-	-
	$V_{1111111011101101}$	-	-	-	-	41	1,630,657
13	$V_{1222233112223231}$	38	4,689	-	-	-	-
	$V_{1222233312223232}$	-	-	38	12,190	-	-
	$V_{1111111111101101}$	-	-	-	-	40	3,465,960
14	$V_{1222211112221211}$	37	18,318	-	-	-	-
	$V_{1222211112221212}$	-	-	37	13,734	-	-
	$V_{1111111111111101}$	-	-	-	-	38	4,810,387
15	$V_{1111111111111111}$	36	7,960,961	36	7,960,961	36	7,960,961

Table 2: Memory occupancy and worst-case execution time.

Bibliography

Aiguier, M., C. Gaston, P. Le Gall, D. Longuet and A. Touil (2005), A temporal logic for input output symbolic transition systems, in *Software Engineering Conference, 2005. APSEC '05. 12th Asia-Pacific*, pp. 8 pp.–, ISSN 1530-1362, doi:10.1109/APSEC.2005.19.

Bagnato, A., L. S. Indrusiak, I. R. Quadri and M. Rossi (2014), *Handbook of Research on Embedded Systems Design*, Hershey, PA: IGI Global, 1st edition, ISBN 978-1-4666-6194-3, doi:10.4018.

Bang-Jensen, J. and G. Gutin (2008), *Digraphs: Theory, Algorithms and Applications*, Springer Publishing Company, Incorporated, 2nd edition, ISBN 1848009976, 9781848009974.

Bergstra, J. and J. Klop (1989), ACPτ a universal axiom system for process specification, in *Algebraic Methods: Theory, Tools and Applications*, volume 394 of *Lecture Notes in Computer Science*, Eds. M. Wirsing and J. Bergstra, Springer Berlin Heidelberg, pp. 445–463, ISBN 978-3-540-51698-9, doi:10.1007/BFb0015048.

Bernstein, P. A., V. Hadzilacos and N. Goodman (1987), *Concurrency Control and Recovery in Database Systems.*, Addison-Wesley, ISBN 0-201-10715-5.

Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2011), LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework, in *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, Eds. P. H. Welch, A. T. Sampson, J. B. Pedersen, J. M. Kerridge, J. F. Broenink and F. R. M. Barnes, IOS Press BV, Amsterdam, pp. 157–175, ISBN 978-1-60750-773-4, ISSN 1383-7575, doi:10.3233/978-1-60750-774-1-157.

Bezemer, M. M., R. J. W. Wilterdink and J. F. Broenink (2012), Design and Use of CSP Meta-Model for Embedded Control Software Development, in *Proceedings of the 34th WoTUG Technical Meeting, Dundee, United Kingdom*, volume 69 of *Concurrent System Engineering Series*, Eds. P. H. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen and A. T. Sampson, Open Channel Publishing Ltd., England, pp. 185–199, ISBN 0-9565409-5-3, ISBN-13 978-0-9565409-5-9.

Birkhoff, G. (1984), *Lattice Theory*, American Mathematical Society colloquium publications, American Mathematical Society, ISBN 9780821810255.

Boehm, B. W. (1984), Verifying and validating software requirements and design specifications, *IEEE Software*, pp. 75–88.

Bollella, G. (2000), *The Real-time Specification for Java*, Addison-Wesley Java Series, Addison-Wesley, ISBN 9780201703238.

Bondy, J. and U. Murty (2008), *Graph Theory*, Springer, Berlin.

Boode, A. and H. Broersma (submitted), Decompositions of graphs based on a new graph product, *Discrete Applied Mathematics*.

Boode, A. H. and J. F. Broenink (2014), Performance of Periodic Real-Time Processes: a Vertex-Removing Synchronised Graph Product, in *Communicating Process Architectures 2014, Oxford, UK*, Open Channel Publishing Ltd, Bicester, 36th WoTUG conference on concurrent and parallel programming, pp. 119–138, ISBN-13 978-0-9565409-7-3.

Boode, A. H. and J. F. Broenink (2016), Asynchronous Readers and Writers, in *Communicating Process Architectures 2016, Copenhagen, Denmark*, Open Channel Publishing Ltd, Bicester, 38th WoTUG conference on concurrent and parallel programming, pp. 125–137.

Boode, A. H. and J. F. Broenink (2017), Asynchronous Readers and Asynchronous Writers, in *Communicating Process Architectures 2017, Valetta, Malta*, Open Channel Publishing Ltd, Bicester, 39th WoTUG conference on concurrent and parallel programming, pp. 79–98.

Boode, A. H., H. J. Broersma and J. F. Broenink (2013), Improving the performance of periodic real-time processes: a graph-theoretical approach, in *Communicating Process Architectures 2013, Edinburgh, UK*, Open Channel Publishing Ltd, Bicester, 35th WoTUG conference on concurrent and parallel programming, pp. 57–79, ISBN-13 978-0-9565409-7-3.

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

Bran, S. and S. Gérard (2014), *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*, Elsevier (Morgan Kaufmann), Amsterdam, ISBN 978-0-12-416619-6.
http://www.sciencedirect.com/science/book/9780124166196

Buttazzo, G. C. (2004), *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*, Springer-Verlag TELOS, Santa Clara, CA, USA, ISBN 0387231374.

Caucal, D. and S. Hassen (2008), Synchronization of grammars, in *Proceedings of the 3rd international conference on Computer science: theory and applications*, Springer-Verlag, Berlin, Heidelberg, CSR'08, pp. 110–121, ISBN 3-540-79708-4, 978-3-540-79708-1.

Cohn, M., S. Even, K. Menger and P. K. Hooper (1962), On the Number of Partitionings of a Set of n Distinct Objects, *The American Mathematical Monthly*, **vol. 69**, pp. 782–785, ISSN 00029890, 19300972.

Comtet, L. (1974), *Advanced Combinatorics: The Art of Finite and Infinite Expansions*, Springer Netherlands, ISBN 9789027704412.

de Boer, M. (2016), Implementation of Periodic Hard Real-Time Processes.

Douglass, B. P. (2014), *Real-Time UML Workshop for Embedded Systems, Second Edition*, Newnes, Newton, MA, USA, 2nd edition, ISBN 0124077811, 9780124077812.

Erdős, P. L. (1993), A new bijection on rooted forests, *Discrete Mathematics*, **vol. 111**, pp. 179 – 188, ISSN 0012-365X, doi:http://dx.doi.org/10.1016/0012-365X(93)90154-L.

FDR (2016), *Failures-Divergence Refinement*, Formal Systems (Europe) Ltd.

Franklin, G. F., D. J. Powell and A. Emami-Naeini (2001), *Feedback Control of Dynamic Systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 4th edition, ISBN 0130323934.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1994), *Design patterns: elements of reusable object-oriented software*, Pearson Education.

Gomaa, H. (2000), *Designing Concurrent, Distributed, and Real-Time Applications with Uml*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, ISBN 0201657937.

Groothuis, M. A., R. M. W. Frijns, J. P. M. Voeten and J. F. Broenink (2009), Concurrent Design of Embedded Control Software, in *Proceedings of the 3rd International Workshop on Multi-Paradigm Modeling (MPM2009), Denver, United States*, volume 21 of *Electronic Communications of the EASST*, Eds. T. Margaria, J. Padberg, G. Taentzer, T. Levendovszky, L. Lengyel, G. Karsai and C. Hardebolle, EASST, Berlin, p. 10, ISSN 1863-2122.

Gruner, S., D. G. Kourie, M. Roggenbach, T. Strauss and B. W. Watson (2008), A New CSP Operator for Optional Parallelism., in *CSSE (2)*, IEEE Computer Society, pp. 788–791, 978-0-7695-3336-0.

Hammack, R., W. Imrich and S. Klavžar (2011), *Handbook of product graphs*, Discrete Mathematics and its Applications (Boca Raton), CRC Press, Boca Raton, FL, second edition, with a foreword by Peter Winkler.

Hammal, Y. (2007), A Component-based Approach for Consistency Checking of UML Dynamic Diagrams, in *(591-193) Software Engineering and Applications - 2007*, volume 591, Ed. J. Smith, Software Engineering and Applications, ACTA PRESS.

Heemels and Muller (Ed.) (2006), *Boderc, Beyond the Ordinary: Design of Embedded Real-time Control*, Embedded Systems Institute.

Hell, P. and J. Nešetřil (2004), *Graphs and Homomorphisms*, Oxford Lecture Series in Mathematics and Its Applications, OUP Oxford, ISBN 9780198528173.

Hoare, C. A. (1985), *Communicating Sequential Processes*, Prentice Hall, 1st edition.

Hoare, C. A. R. (1978), Communicating Sequential Processes, *Commun. ACM*, **vol. 21**, pp. 666–677, ISSN 0001-0782, doi:10.1145/359576.359585.

Imrich, W. and H. Izbicki (1975), Associative products of graphs, *Monatsh. Math.*, **vol. 80**, pp. 277–281.

ISO (1987), ISO - Information Processing Systems - Open Systems Interconnection - "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", DIS 8807.

Klapuri, H., J. Takala and J. Saarinen (1999), Safety, liveness and real-time in embedded system design, *Journal of Network and Computer Applications*, **vol. 22**, pp. 69 – 89, ISSN 1084-8045, doi:http://dx.doi.org/10.1006/jnca.1999.0083. http://www.sciencedirect.com/science/article/pii/S1084804599900838

Kopetz, H. (1997), *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, Norwell, MA, USA, 1st edition, ISBN 0792398947.

Lamport, L. (2002), *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 032114306X.

Li, C., C. Ding and K. Shen (2007), Quantifying the cost of context switch, in *Proceedings of the 2007 workshop on Experimental computer science*, ACM, New York, NY, USA, ExpCS '07, ISBN 978-1-59593-751-3, doi:10.1145/1281700.1281702.

Lomet, D. B. (1976), *Process structuring synchronisation and recovery using atomic actions*, Newcastle upon Tyne: University, Computing Laboratory.

Magee, J. and J. Kramer (1999), *Concurrency: State Models &Amp; Java Programs*, John Wiley & Sons, Inc., New York, NY, USA, ISBN 0-471-98710-7.

Marwedel, P. (2010), *Embedded system design: Embedded systems foundations of cyber-physical systems*, Springer Science & Business Media.

Milner, R. (1989), *Communication and Concurrency*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, ISBN 0-13-115007-3.

Nakata, K. and T. Uustalu (2009), Trace-based coinductive operational semantics for while, in *Theorem Proving in Higher Order Logics*, Springer, pp. 375–390.

Object Management Group (OMG) (2015), UML Specification, Version 2.5.

Oguz, O., J. Broenink and A. H. Mader (2012), Schedulability Analysis of Timed CSP Models Using the PAT Model Checker, in *Communicating Process Architectures 2012*, volume 69 of *Concurrent System Engineering Series*, Eds. P. H. Welch, F. R. M. Barnes, K. Chalmers, J. B. Pedersen and A. T. Sampson, Open Channel Publishing, pp. 65–88, ISBN 978-0-9565409-5-9.

Pinto, R. S., F. J. Monaco, J. C. Faracco and J. R. Monteiro (2012), Embedded Linux in Real-Time Applications: Performance Enhancements of Experimental Fully-Preemptible Capabilities over the Standard Kernel in a Critical Mobile System, in *Critical Embedded Systems (CBSEC), 2012 Second Brazilian Conference on*, IEEE, pp. 76–81.

Roscoe, A. W. (1998), *The theory and practice of concurrency*, Prentice Hall, ISBN 0-13-6774409-5.

Roscoe, A. W. (2010), *Understanding Concurrent Systems*, Springer, ISBN 978-1-84882-257-3.

Sabidussi, G. (1960), Graph multiplication, *Mathematische Zeitschrift*, **vol. 72**,

pp. 446–457, ISSN 0025-5874, doi:10.1007/BF01162967.

Scattergood, B. (1998), *The Semantics and Implementation of Machine-Readable CSP*, Ph.D. thesis, University of Oxford.

Schneider, S. (1999), *Concurrent and Real Time Systems: The CSP Approach*, John Wiley Sons, Inc., New York, NY, USA, chapter 1, p. 1, 1st edition, ISBN 0471623733.

Schäfer, T., A. Knapp and S. Merz (2001), Model Checking UML State Machines and Collaborations, *Electronic Notes in Theoretical Computer Science*, **vol. 55**, pp. 357 – 369, ISSN 1571-0661, doi:http://dx.doi.org/10.1016/S1571-0661(04)00262-2.
http://www.sciencedirect.com/science/article/pii/S1571066104002622

Sha, L., R. Rajkumar and J. P. Lehoczky (1990), Priority Inheritance Protocols: An Approach to Real-Time Synchronization, *IEEE Trans. Comput.*, **vol. 39**, pp. 1175–1185, ISSN 0018-9340, doi:10.1109/12.57058.
http://dx.doi.org/10.1109/12.57058

Spanoudakis, G. and A. Zisman (2001), Inconsistency management in software engineering: Survey and open research issues, *Handbook of software engineering and knowledge engineering*, **vol. 1**, pp. 329–380.

Tanenbaum, A. S. and A. S. Woodhull (2005), *Operating Systems Design and Implementation (3rd Edition)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, ISBN 0131429388.

Veldhuijzen, B. (2009), *Redesign of the CSP execution engine*, MSc thesis 036CE2008, Control Engineering, University of Twente.

Vrancken, J. L. M. (1997), The Algebra of Communicating Processes with Empty Process, *Theor. Comput. Sci.*, **vol. 177**, pp. 287–328, ISSN 0304–3975.

Welch, P. H. and J. M. R. Martin (2000), A CSP Model for Java Multithreading, in *International Symposium on Software Engineering for Parallel and Distributed Systems, PDSE 2000, Limerick, Ireland, June 10-11, 2000*, pp. 114–122, doi:10.1109/PDSE.2000.847856.
http://dx.doi.org/10.1109/PDSE.2000.847856

Wellings, A. (2004), *Concurrent and Real-Time Programming in Java*, John Wiley & Sons, ISBN 047084437X.

Wöhrle, S. and W. Thomas (2004), Model checking synchronized products of infinite transition systems, in *in: Proc. 19th LICS, IEEE Comp. Soc*, IEEE Computer Society Press, pp. 2–11.

Index