Master Thesis

Synthesis-driven Derivation of Process Graphs from Functional Blocks for Time-Triggered Embedded Systems

by

Ghennadii Sivatki

LITH-IDA/DS-EX--05/010--SE

2005-11-25

Linköpings universitet
Department of Computer and Information Science

Master Thesis

Synthesis-driven Derivation of Process Graphs from Functional Blocks for Time-Triggered Embedded Systems

by

Ghennadii Sivatki

LITH-IDA/DS-EX--05/010--SE

2005-11-25

Supervisor: Dr. Paul Pop

Examiner: Dr. Paul Pop

Таисии и Григорию

To
Taisia and Grigoriy

Abstract

Embedded computer systems are used as control systems in many products, such as VCRs, digital cameras, washing machines, automobiles, airplanes, etc. As the complexity of embedded applications grows and time-to-market of the products they are used in reduces, designing reliable systems satisfying multiple requirements is a great challenge. Successful design, nowadays, cannot be performed without good design tools based on powerful design methodologies. These tools should explore different design alternatives to find the best one and do that at high abstraction levels to manage the complexity and reduce the design time.

A design is specified using models. Different models are used at different design stages and abstraction levels. For example, the functionality of an application can be specified using hierarchical functional blocks. However, for such design tasks as mapping and scheduling, a lower-level flat model of interacting processes is needed. Deriving this model from a higher-level model of functional blocks is the main focus of this thesis. Our objective is to develop efficient strategies for such derivations, aiming at producing a process graph specification, which helps the synthesis tasks to find schedulable implementations. We proposed several strategies and evaluated them experimentally.

Acknowledgements

I would like to thank my supervisor, Dr. Paul Pop, for the interesting idea of this thesis, the opportunity to work on this project, and useful discussions.

I am so deeply grateful to my loving family, Taisia, Grigoriy, Vladimir and Alexei, and to my dearest friend Tatiana who have always been there for me, in spite of the geographical distance, with their support, understanding, encouragement and belief in me.

Contents

List of Figures

List of Tables

List of Abbreviations

AGPD	Allocation Group to Process Directly
AGPES	Allocation Group to Process based on Exhaustive Search
AGPSA	Allocation Group to Process based on Simulated Annealing
AGPSD	Allocation Group to Process based on Steepest Descent
AGPTS	Allocation Group to Process based on Tabu Search
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
CNI	Communication Network Interface
CPU	Central Processing Unit
DSP	Digital Signal Processor
EDF	Earliest Deadline First
EFP	Elementary Function to Process
ET	Event-Triggered
FPGA	Field Programmable Gate Array
FPS	Fixed Priority Scheduling
FSM	Finite State Machine
I/O	Input/Output
IP	Intellectual Property
MBI	Message Base Interface
MEDL	Message Descriptor List
NP	Non Polynomial
PCB	Process Control Block
RAM	Random Access Memory
RG	Restricted Growth
ROM	Read-Only Memory
SA	Simulated Annealing
TDMA	Time-Division Multiple Access
TS	Tabu Search
TT	Time-Triggered
TTP	Time-Triggered Protocol
WCAO	Worst-Case Administrative Overhead
WCEO	Worst-Case Execution Overhead
WCET	Worst-Case Execution Time
WCTT	Worst-Case Transmission Time

1 Introduction

Embedded systems is a huge class of computer systems that are used to control the operation of larger host systems in which they are embedded. The examples are television sets, microwave ovens, mobile phones, wristwatches, washing machines, etc. Most embedded systems are real-time computer systems, which means that they operate under strict timing constraints. Many such systems are used in safety-critical applications. In addition to functional and timing, embedded systems must satisfy other requirements (power consumption, cost, etc.) that often compete with each other. Besides that, the highly competitive market of devices embedded systems are used in puts additional pressure on designers in the form of the time-to-market requirement. In order for a product to succeed in the market, this time should be reduced as much as possible. All these make the design of embedded systems very difficult. To manage the difficulty, it is extremely important to have powerful design methodologies and tools that would free the designer of low-level decisions and tasks.

Design of embedded systems is usually an iterative process that explores different design alternatives and chooses the one that satisfies all the requirements. A variety of models can be used during the design. They describe the functionality of the application or its structure at different abstraction levels. Refinement and synthesis of models is necessary for the design space exploration. For safety-critical systems, which are implemented as time-triggered systems, the main model is a model of communicating processes that are assigned for execution (mapped) to computational resources of the system architecture and scheduled. This model must satisfy all the requirements to the system. Synthesising such a model from a higher-level specification of functionality is the topic of this thesis. The aim is to develop strategies for derivation of mapped and scheduled system models from higher-level behavioural system models.

1.1 Thesis overview

This thesis is structured in eight chapters. Chap. 2 gives an introduction to real-time and embedded systems, and their design. Chap. 3 describes, states and motivates the problem of the thesis. In Chap. 4, we dwell on a particular kind of embedded systems, which are the main focus of the thesis, time-triggered embedded systems. We describe the hardware and software architectures that we consider. In Chap. 5, we propose strategies for derivation of system models represented as process graphs from the ones represented as functional blocks. Chap. 6 presents the results achieved during experimentation with the implemented derivation strategies. Chap. 7 describes related work. And in Chap. 8, we conclude and propose ideas for future work.

2 Technical Background: Embedded Systems

The purpose of this chapter is to give an introduction to the subject area of this thesis that is of embedded computer systems, to define their key concepts, and describe their design methodologies.

2.1 Real-Time Systems

Most of embedded computer systems are real-time computer systems, so are those this thesis is aiming at. Therefore, first of all, we will define what a real-time system is.

"A computer system in which the correctness of the system behaviour depends not only on the logical results of the computations, but also on the physical instant at which these results are produced" is called a real-time computer system [8].

A real-time computer system always has an environment, in which it operates. The environment consists of a *human operator* and a *controlled object* with corresponding interfaces – a *man-machine interface* (MMI) and an *instrumentation interface* (II) (Fig. 2.1). The man-machine interface includes input and output devices (e.g., keyboard and display). The instrumentation interface includes a set of sensors and actuators, which transform physical signals of the controlled object into a digital form understandable by the computer system and vice versa. The combination of a real-time computer system and its environment is called a *real-time system* [8].

Examples of real-time systems are engine control systems in vehicles, ticket-booking systems, automatic teller machines, plant automation systems, flight control systems, etc.

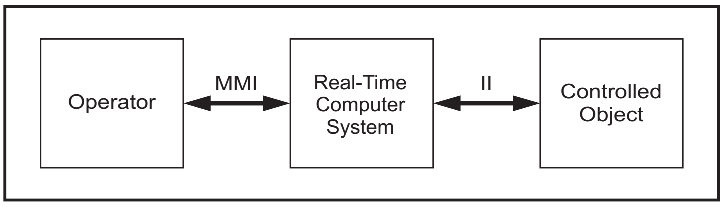


Fig. 2.1. Real-time system (adapted from [8])

A real-time computer system must react to stimuli from its environment within time limits imposed by the environment. The instant of time at which a result must be produced is called a *deadline*. If a result is still useful after the deadline has passed, the deadline is called *soft*, if not, the deadline is *firm*. If missing a firm deadline may result in a catastrophe, the deadline is called *hard*. A real-time computer system that must meet at least one hard deadline is classified as a *hard real-time computer system* or a *safety-critical real-time computer system* [8]. For example, for a nuclear plant control system, failure to react at appropriate time may lead to catastrophic consequences.

If a real-time computer system does not have any hard deadlines, then it is called a *soft real-time computer system* [8]. For example, a multimedia system may occasionally miss its deadlines, which will result in a degraded quality of image and sound, which in turn will cause only inconvenience to the user.

There are several viewpoints real-time systems can be classified from [8].

- Hard real-time versus soft real-time,
- Fail-safe versus fail-operational,
- Guaranteed-timeliness versus best-effort,
- Resource-adequate versus resource-inadequate, and
- Event-triggered versus time-triggered.

The first two classifications depend on the characteristics of the application, i.e., the controlled object. The rest depend on the implementation of the computer system.

Hard real-time systems are different from the soft ones in response time, peak-load performance, control of pace, safety, size of data files, and redundancy type.

Fail-safe and *fail-operational* systems are different in the way they behave upon occurrence of a failure. The systems, for which safe states exist, transition to one of the states in case of a failure. The systems, for which a safe state cannot be identified, remain operational, possibly providing a limited service.

Guaranteed-response systems are designed in such a way that they guarantee a response in the case of a peak load and fault scenario with the probability that the assumptions about the peak load and faults made during the design will hold in reality. *Best-effort* systems do not give such an analytic guarantee; instead, they establish the sufficiency of the design during testing and integration.

Resource-adequate and *resource-inadequate* systems differ in availability of resources under peak load and fault scenarios. Guaranteed response requires resource adequacy. Hard real-time systems must be designed according to the guaranteed response paradigm.

Event-triggered and *time-triggered* systems are different in the triggering mechanism used to start an action (e.g., execution of a task or sending a message) in a node of a computer system.

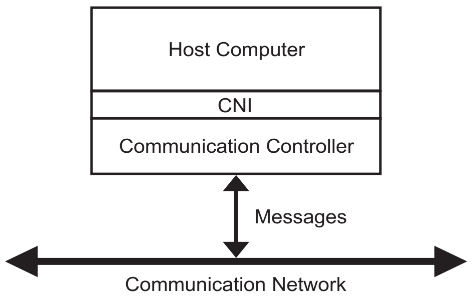


Fig. 2.2. Structure of a node in a distributed real-time system (adapted from [8])

2.2 Distributed Real-Time Systems

Many real-time applications are distributed by their nature. For example, plant automation systems, or automotive electronics. In such applications, it makes sense to perform the data processing at the location of the controlled objects. In this case, the controlling computer system becomes a *distributed real-time computer system*. It is implemented as a set of hardware components, called *nodes*, interconnected by a real-time communication network. Many real-time systems are designed as distributed systems due to fault tolerance or performance issues.

A node of a distributed real-time system can be structured as two subsystems (Fig. 2.2), the host computer, and the communication controller interfacing with each other through the *communication network interface* (CNI) [8].

The host computer contains a central processing unit (CPU), a memory, and a real-time clock that is synchronized with the real-time clocks of all the other nodes. The memory stores the node's software – the real-time operating system, input/output drivers, and the application programmes.

The set of all the communication controllers of the nodes and the physical interconnection medium form the real-time communication system of a distributed real-time system [8].

2.3 Embedded Real-Time Systems

Due to constantly decreasing price-performance ratio of microcontrollers, it has become economically attractive to replace the conventional mechanical or electronic control systems inside many products by real-time computer systems. In this case, such a computer system is called an *embedded computer system* [8]. The examples of products with embedded computer systems are telephones, watches, electronic toys, digital cameras, television sets, computer printers, washing machines, microwave ovens, automobiles, airplanes, etc. More than 99% of the microprocessors produced today are used in embedded systems [12].

Although the number and diversity of embedded systems is huge, they share a small set of common characteristics:

- Since such computer systems are embedded into a host system, they are designed to perform a dedicated set of functions determined by the purpose the host system serves to.
- Besides correct functionality under timing constraints, the design of embedded systems has to consider many other tight, varied, and competing issues, such as development cost, unit cost, size, power consumption, flexibility, time-to-prototype, time-to-market, maintainability, safety, etc.
- Both hardware and software aspects, and under the above constraints, have to be considered simultaneously during the design of embedded systems. That makes the design very difficult.

2.4 Real-Time Scheduling

At run time, the functionality of a real-time system can be represented as a set of concurrent *processes* (*tasks*) executed by the system's computational resources (e.g., programmable processors, application specific integrated circuits, etc.).

A process is a sequence of computations, which starts when all its inputs are available. When it finishes executing, the process produces its output values [9]. The control signal that initiates the execution of a process is provided by the operating system. Important attributes of a process are:

- *Release (arrival*, or *request*) *time*, the time when the process becomes ready for execution,
- *Execution time*, the time it takes to execute the process,
- *Worst-case execution time* (*WCET*), *C*, the maximal time it may take to execute the process,
- *Period*, *T*, the time interval between successive release times,
- *Deadline*, *D*, the time by which the execution of a process must be finished,
- *Priority*, π, the level of importance of the process.

Based on the release times, processes can be:

- *Periodic*, when the release times are known and the intervals between them are constant,
- *Aperiodic*, when the release times are not known, and
- *Sporadic*, they are aperiodic processes for which minimal inter-release times are known.

A real-time system must execute the set of its concurrent processes in such a way that all processes meet their deadlines. Then, the scheduling problem is which process and at what moment has to be executed on a given computational resource in order for all timing requirements to be satisfied. The following classifications of scheduling approaches exist:

- Static versus dynamic,
- Off-line versus on-line, and
- Preemptive versus non-preemptive.

Scheduling is *static* if pre-run-time predictions about the worst-case behaviour of the system when the scheduling algorithm is applied are made. Examples of the static approach are cyclic scheduling and fixed priority scheduling (FPS). If no pre-run-time predictions are made but instead run-time decisions are used, the scheduling is *dynamic*. An example of the dynamic approach is earliest deadline first (EDF) scheduling.

When there is complete a priori knowledge about the process-set characteristics, e.g. release times, WCETs, precedence relations, etc., a schedule table for the run-time scheduler can be generated *off-line*. Such a table contains all the information needed to decide at run time which task is to be dispatched next. An example of the off-line scheduling approach is static cyclic scheduling. If there is no such complete a priori knowledge, the scheduling decisions have to be made at run time (*on-line*). Examples of the on-line scheduling approach are FPS and EDF scheduling.

In *preemptive* scheduling, the currently executing task may be preempted (interrupted), if a task with a higher priority has been released. In *nonpreemptive* scheduling, the currently executing task may not be interrupted.

In order to determine whether a process set is schedulable, that is it can be scheduled so that each task meets its deadline, or not, a *schedulability analysis* (*test*) is performed. Schedulability analysis can be based on sufficient, necessary, or necessary and sufficient conditions. If the sufficient condition is satisfied, the process set is definitely schedulable; otherwise, the process set may be or not be schedulable. If the necessary condition is not satisfied, the process set is definitely not schedulable; otherwise, the set may be or not be schedulable. Tests based on both necessary and sufficient conditions give an exact answer at the expense of their complexity. Tests based either on sufficient or on necessary conditions may give either too pessimistic or too optimistic results respectively, but they are simpler. For the off-line scheduling approach, construction of a schedule is considered a sufficient schedulability test.

2.5 Embedded Systems' Design Flow

Design of embedded systems is a difficult task due to the following reasons:

- It has to deal with the very complex functionality of modern applications. Many real-time applications have to be implemented as multiprocessor systems with heterogeneous structure. This means they contain both programmable processors and dedicated hardware components. For example, the programmable

processors can be general purpose or application specific microprocessors, microcontrollers, digital signal processors (DSPs). The dedicated hardware components can be implemented as application specific integrated circuits (ASICs) or field programmable gate arrays (FPGAs). In addition to that, for distributed systems, the network structure can be heterogeneous too. It can consist of several networks, interconnected with each other, with each network having its own communication protocol,
- Both hardware and software have to be designed simultaneously,
- In addition to desired functionality and correct timing, it has to take into consideration many tight, and often competing requirements, such as development cost, unit cost, size, power consumption, flexibility, time-to-prototype, time-to-market, maintainability, safety, etc.

In order to simplify the design process, to reduce its cost and time-to-market, and to address the complexity of applications, it is very important to have:

- Powerful design methodologies, which would perform as many of the activities considering different design alternatives as possible at high abstraction levels,
- Efficient design tools based on the above methodologies, which would allow to automate most of the design tasks, ideally the whole design process,
- Reuse of previously designed and verified hardware and software blocks, and
- Good designers.

Design of embedded systems can be performed at different abstraction levels, with different types of models used at each level, behavioural and structural. A behavioural model describes the desired functionality of the system, and a structural model describes the composition of the system of physical elements. For example, the hardware can be designed at:

- *Circuit level*, the lowest abstraction level. The main building blocks here are transistors. The system's functionality is described using differential equations,
- *Logic level*. With building blocks being gates and flip-flops and the functionality described as Boolean logic,
- *Register-transfer level*. Where transfers of values between registers, multiplexers, ALUs, etc. describe the functionality,
- *System level*, the highest abstraction level. Here, the structure is captured using processors, memories, ASICs, and communication channels and the functionality is described using system level specification formalisms (e.g., hardware description languages).

It is the system level the design methodologies should make most of the design space exploration at.

Fig. 2.3 shows such a design flow. It was suggested by researchers to systematically address the design of embedded systems [4].

The activities undertaken during the design based on this methodology are the following:

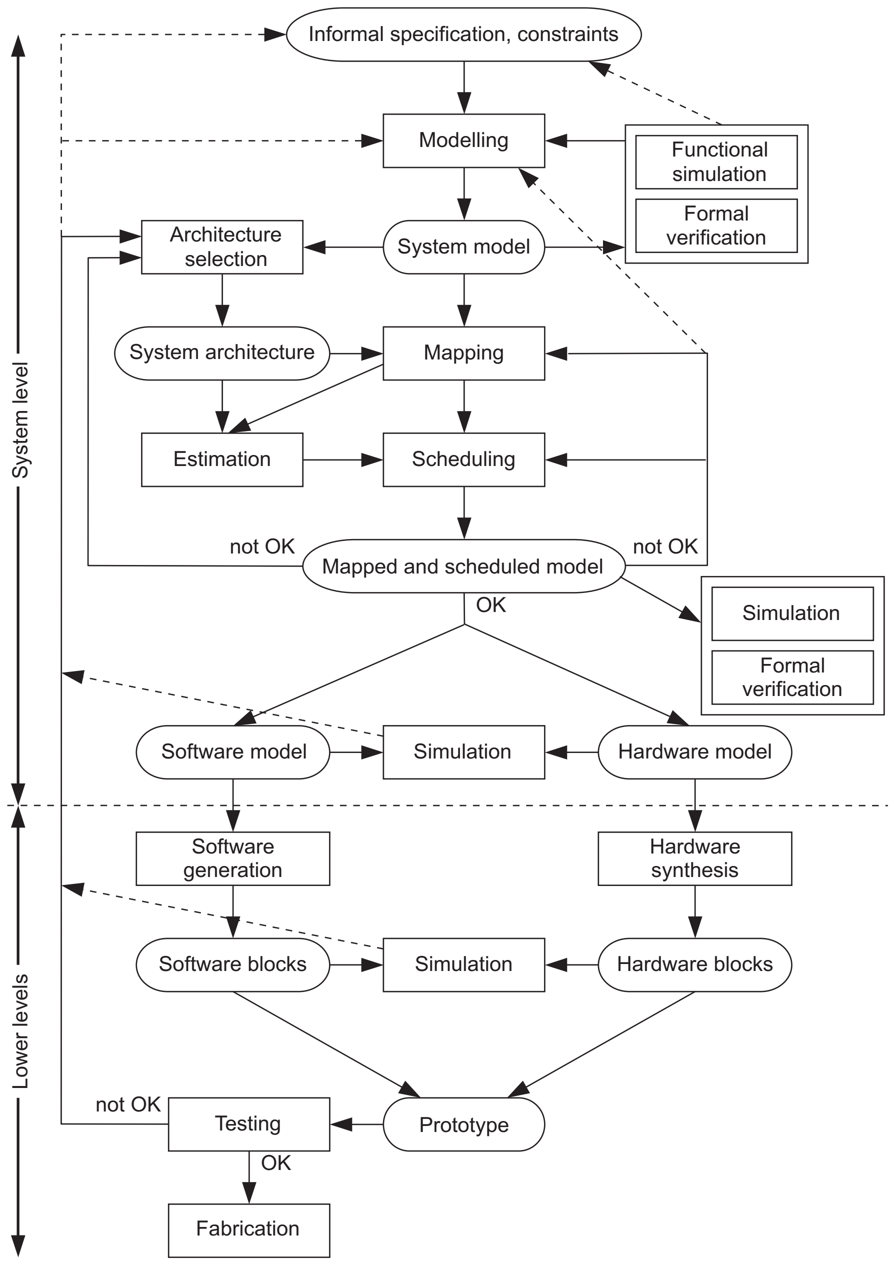


Fig. 2.3. Embedded systems' design flow [4]

1. The flow starts from the informal specification (e.g., natural language) of the functionality and the set of constrains (e.g., power consumption, cost, etc.).
2. As a result of modelling, the system model is produced, possibly using several refinement steps. The system model is a formal specification of the functionality, based on a modelling concept (e.g. finite state machine, data-flow, Petri net, etc.).
3. At this step, the system's architecture (processors, custom intellectual property (IP) modules, busses, operating system, etc.) is chosen.
4. Mapping of the system's functionality on the computational resources is performed. This means to choose a resource where a given piece of functionality is going to be executed in. Here, the decision about what will be implemented as hardware and what will be implemented as software is made.
5. Next, based on the results of the mapping, on the estimation of the WCETs of the process-set with the given mapping, and on the chosen scheduling algorithm, schedulability analysis is performed. If the static cyclic scheduling was chosen (the case in Fig. 2.3), then the scheduling tables for each processing element are built. Depending on the result, the flow may move forward or go back to the previous stages to explore different architectures, mappings, and schedules.
6. Now, the partitioning of the system model into the software model and the hardware model can be done. The software model describes the functionality mapped to processors, and the hardware model describes the functionality mapped to ASICs and FPGAs.
7. And finally, the low-level design tasks, which are automatic or manual software generation and hardware synthesis, prototyping, and testing the prototype, are performed.

At all the stages of the design, the model should be either formally verified or simulated to check its validity. Depending on the results of validation, adjustments to the model can be made.

The described approach is a so-called *function/architecture co-design* methodology. The main characteristic of this methodology is that it uses a top-down synthesis approach and, at the same time, a bottom-up evaluation of design alternatives [9].

Usually a design does not start from scratch. The design is based on a hardware platform, which is instantiated for the given application by parameterizing the platform's components (e.g., the frequencies of the processors, the sizes of the memories, etc.). Such a design is called *platform based*.

3 Problem Description

As we have said in the previous chapter, it is very important for the design of embedded system to have powerful design methodologies that could handle the high complexity and tight requirements of embedded applications. One of such methodologies was described in Sect. 2.5. However, there are issues inside a methodology itself that are essential for a successful design, such as strategies and techniques used for a particular activity of the design. The activity this thesis is in concern of is automatic creation of a mappable system model from a higher-level behavioural specification (see Fig. 2.3).

3.1 System-Level Modelling

There are two models in the design flow shown in Fig. 2.3 denoted as "System model" and "Mapped and scheduled model" we are interested in. The system model is a behavioural model describing the functionality of the application being designed as a set of behaviours that can be directly mapped onto the processing elements of the system architecture. The mapped and scheduled model can be viewed as a structural model where each physical element from the system architecture is assigned a piece of functionality from the system model to execute (in a specified order and at specified time instances[1]). The process of transforming a behavioural representation into a structural representation is known as synthesis. Therefore, we say that the system model is synthesized into the mapped and scheduled model during the design.

The model used for mapping and scheduling is a flat model of interacting behaviours that are represented as processes at run-time. For complex applications, it may be very difficult to construct such a model directly from the informal specification of functionality. Therefore, a move to higher abstraction levels in formal behavioural descriptions is necessary. Then, the transition from the informal specification to the model of interacting processes can be done as a sequence of refinement[2] steps.

A possible modelling scenario could be the following. First of all, based on the requirements and constraints of the informal specification, an overall behavioural

[1] We assume that a static cyclic scheduling is used, which allows building scheduling tables off-line.

[2] *Refinement* is a process of lowering the abstraction level of a model by adding more details.

model of the system is constructed as a single black box transforming its inputs to the outputs. This model can be built using a system level modelling formalism (suitable for the given application), such as a Petri Net or a Finite State Machine (FSM), which is mathematically verifiable. Next, the overall behaviour is decomposed into a set of interacting concurrent hierarchical behaviours. The decomposition is done following, for instance, availability of reusable hardware and software IP components, implementation constraints, or the designer's vision of how it should be done. And finally, the model of hierarchical behavioural blocks is refined into a model suitable for processing by the synthesis task.

This last refinement is a problem we are going to solve.

In the following two subsections, we will describe the representations that will be used for the two models that are an input and an output of the refinement problem.

3.1.1 Functional Blocks

In our representation, a behaviour in the model of interacting concurrent hierarchical behaviours is called a *functional block*, or a *function*. Each function may consist of other functions. We call the leaf functions *elementary*. *Composite functions* are the functions that include elementary or other composite functions.

The model is represented as a directed hierarchical graph $H = (F_e, F_c, D, h, c)$, where F_e is a set of elementary functions; F_c is a set of composite functions; $F_e \cup F_c = F$, where F is a set of all functions; D is a set of edges between the elements of F_e; h: $F_c \rightarrow F_c$ is an inclusion relation between the composite functions; c: $F_e \rightarrow F_c$ is an inclusion relation between the elementary and composite functions. An element of D is called a *dependence* and represents data dependency between a pair of functions. This graph is a kind of a *dataflow graph*.

An elementary function is an implementable behaviour (the implementation can be done in a modelling or a programming language). A composite function is a specifying behaviour, which means that it defines how the functional hierarchy (with this function in the root) should behave.

Given system architecture, timing information for elementary functions and dependences can be specified. For an elementary function, the timing characteristics are its worst-case execution time and period. The worst-case execution time of a function is the maximal time it may take to execute the function on a reference computational resource of the system architecture. The WCETs of all the elementary functions are estimated for the same reference resource. For techniques on WCET estimation, please refer to [3]. The period is the time interval between successive invocations of the function. It is specified in the requirements to the application being modelled.

For a dependence, the timing attribute is its worst-case transmission time. Each dependence is associated with an amount of data, which is passed between the two functions the dependence connects. Hence, its WCTT is the time it takes to transmit those data over a reference bus of the system architecture.

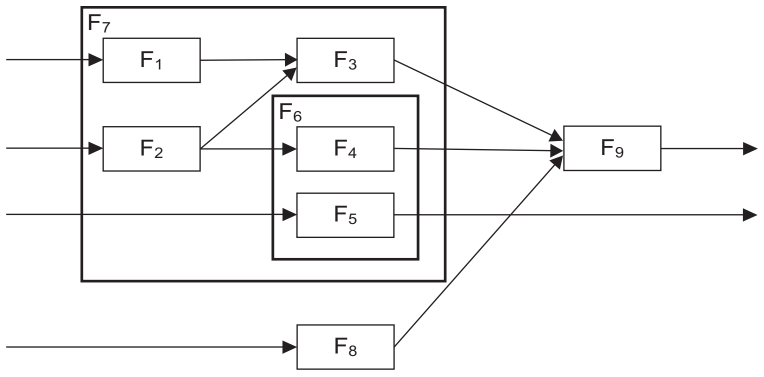


Fig. 3.1. Example of a graph of functional blocks

Example 3.1: In the graph of functional blocks shown in Fig. 3.1, $F_e = \{F_1, F_2, F_3, F_4, F_5, F_8, F_9\}$, and $F_c = \{F_6, F_7\}$.

3.1.2 Process Graphs

We represent the model of interacting processes as a *process graph* (also called a *task graph*), which is a kind of a dataflow graph [2]. A process graph is a directed acyclic graph $G = (P, M)$, where P is a set of processes, and M is a set of edges representing dependences between the processes in the form of messages [9].

The graph is a polar graph, which means that there are two special vertices in it, called *source* and *sink* that represent the first and the last processes respectively. These processes are dummy, they do not perform any computations, their execution time is zero, and they are not assigned to any computational resources. All the other processes in the graph are successors of source and predecessors of sink [9].

A mapped process graph $G^* = (P^*, M^*, m)$ is generated from a process graph $G = (P, M)$ by introducing communication processes and by mapping each process to a processing element from the set of processing elements PE. $PE = R_{comp} \cup R_{comm}$, where R_{comp} is a set of the system's computational resources (i.e., programmable processors and dedicated hardware), and R_{comm} is a set of the systems communicational resources (i.e. busses). Communication is performed using message passing through shared memory if the communicating processes are mapped to the same computational resource, or through a bus if the communicating processes are mapped to different computational resources. Communication processes are inserted in edges that represent messages sent using the busses. Function m: $P^* \rightarrow PE$ gives the mapping of the processes in P to the processing elements in PE. For every process $P_i \in P$, $m(P_i)$ is the processing element to which P_i is assigned for execution [9].

Timing information for a process graph is the same as for a graph of functional blocks – worst-case execution times, periods, and worst-case transmission times. However, for a mapped process graph it can be expressed for those resources the processes and messages were mapped to. In addition, the messages between the processes mapped to the same computational resource are considered to take no time.

Example 3.2: In the mapped process graph in Fig.3.2, P_0 and P_{12} are source and sink processes, respectively. $P_1, P_2, \ldots, P_{11}$ are processes that perform computations. They are assigned for execution to one of the computational resources (Processor 1, Processor 2, or ASIC), as indicated by the shadings. The solid black circles in the figure depict communication processes. They are introduced after the mapping of the computations is done, and only for those communications that are performed between processes mapped to different computational resources. The communication processes are mapped to the single bus. A number to the right of each process is the WCET of that process on the given resource. The worst-case execution time of a communication process is equal to the worst-case transmission time of the message this processes represents.

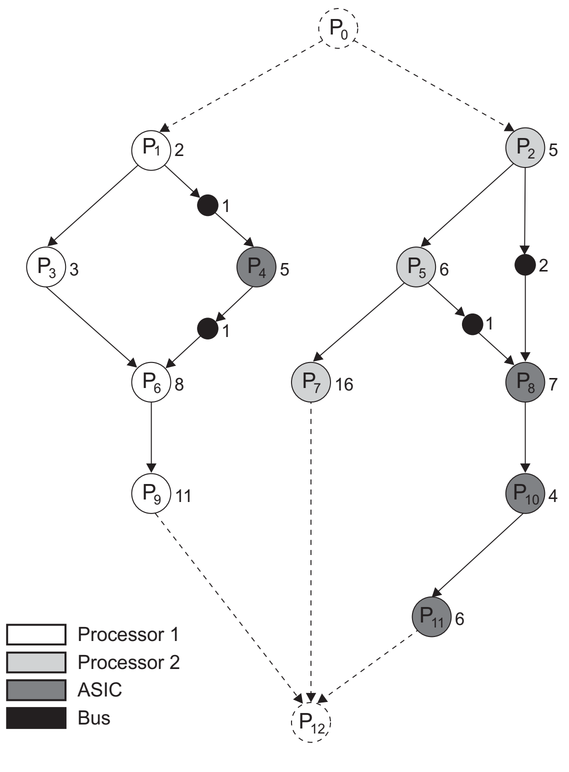


Fig. 3.2. Example of a process graph

3.2 Problem Statement

We state the problem of this thesis as follows.

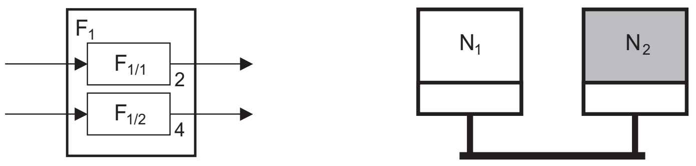
Given a hierarchical behavioural model represented as a graph of functional blocks, system architecture, and a synthesis algorithm, the problem is to build a flat behavioural model represented as a process graph such that the implementation obtained using the synthesis algorithm is schedulable. The synthesis algorithm produces mapping, and schedule tables (since we are interested in time-triggered systems, static cyclic scheduling is used).

The procedure of building a process graph based on the graph of functional blocks consists of building processes out of functional blocks. A process may be constructed out of one or several functions. We say that in this procedure, functional blocks are *allocated* to processes. The goal is to find such an allocation that the resulting process graph is schedulable. The search is driven by the given synthesis algorithm, which maps the process graph under consideration to the given architecture, schedules it, and gives the answer whether the process graph is schedulable or not. Often the designer is interested in finding a process graph that is not only schedulable but also has minimal schedule length. We will use schedule length as a search criterion.

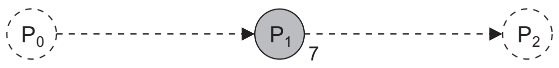
3.3 Motivation

We will motivate the problem by giving several motivational examples.

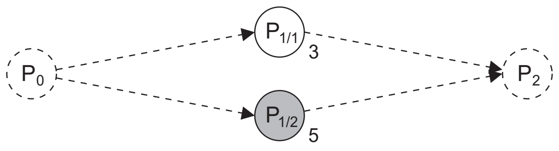
Example 3.3: Let us consider a simple application that is modelled as the graph of functional blocks shown in Fig. 3.3a. It is a composite function F_1 that includes two elementary functions $F_{1/1}$ and $F_{1/2}$. The elementary functions are invocated every 6 time units, and they must finish executing before they are invocated again. The application is to be implemented on the architecture in Fig. 3.3b, consisting of two computational nodes N_1 and N_2, and a bus. The nodes have the same performance, and the execution of the elementary functions on either of them would take 2 time units for $F_{1/1}$ and 4 time units for $F_{1/2}$. There are two ways to transform the given hierarchical graph of functional blocks into a flat process graph. They are shown in Fig. 3.3c, d. In the first case, the process graph consists of a single process P_1 that performs the computations of both elementary functions $F_{1/1}$ and $F_{1/2}$. In the second case, two processes are created, each corresponding to an elementary function. The worst-case execution time of a process is a sum of the WCETs of the elementary functions making up the process. In addition to that, a process's WCET should also include the overhead for executing the process by the operating system at run-time (the overheads will be discussed in Sect. 5.2.1). For this example, we consider the overhead for each process to be equal to 1 time unit. Then the WCETs of the created processes are:



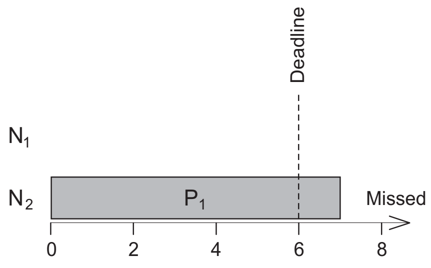
a) Application **b)** Architecture



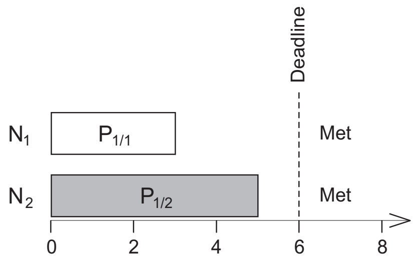
c) Process graph, case 1



d) Process graph, case 2



e) Scheduling, case 1



f) Scheduling, case 2

Fig. 3.3. First motivational example

- In the first case, 7 (the functions have to be executed sequentially inside the process), and
- In the second case, 3 for $P_{1/1}$ and 5 for $P_{1/2}$.

Fig. 3.3c, d also illustrate the mappings of the processes to the computational nodes. With such mappings, scheduling can be performed as shown in Fig. 3.3e, f (the width of the rectangle depicting a process corresponds to the process's execution time). As we can see, the first process graph is unschedulable, while the second solution satisfies all the deadlines (if the processes are mapped to different nodes).

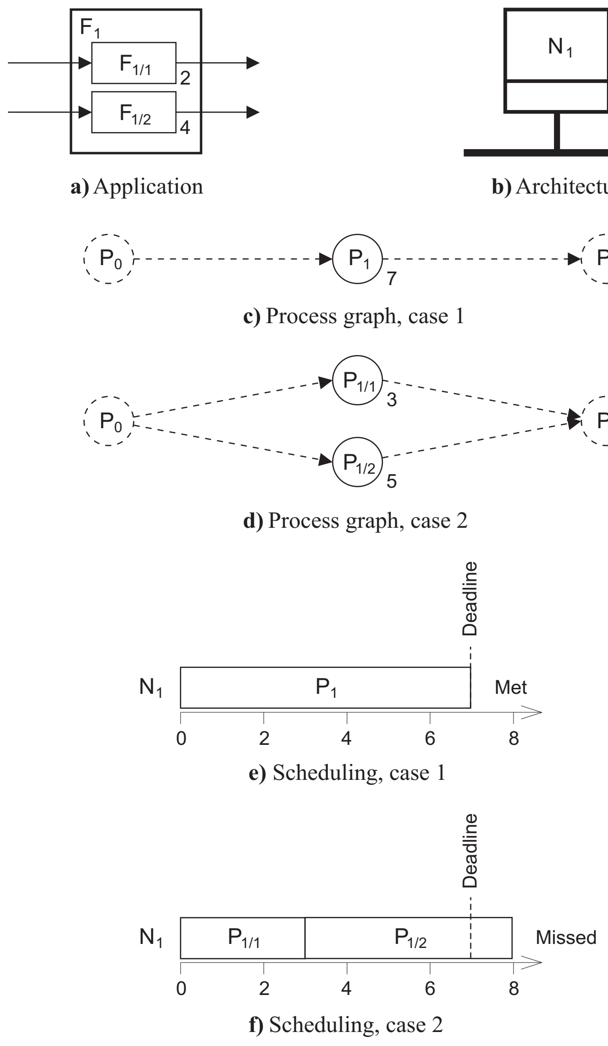


Fig. 3.4. Second motivational example

Example 3.4: Let us consider the same application as in Fig. 3.3. However, the periods (and the deadlines) of the elementary functions equal to 7, and the architecture consists only of one node (Fig. 3.4a, b). Using the same approach to creating process graphs, we get a schedulable solution when the elementary functions are combined into the same process (Fig. 3.4c, e), and an unschedulable solution when a separate process corresponds to each of the elementary functions (Fig. 3.4d, f). The total execution time of the application is larger in the second case because there are two processes and the overhead has to be considered twice.

Example 3.5: Fig. 3.5 and Fig. 3.6 give an example of a more complex application. Here, we assume that sending messages between the processes mapped to different nodes takes zero time since scheduling of messages is not relevant for this example. We also assume that the nodes have equal performance. In the first case, process P_6 does not meet its deadline, and it is not possible to solve this problem by a different mapping because of the chain of dependences $P_1 - P_3 - P_4 - P_6$. However, if each of the elementary functions inside F_3 is represented as a separate process (the second case), all the deadlines are met.

It is clear from the above examples that the way processes are built out of functional blocks affects the timing of an application and leads to either schedulable or not schedulable design. For complex graphs of functional blocks with large number of functions, multiple hierarchical levels, and constraints on function allocation, creating a schedulable flat process graph is not as straightforward as in the simple examples we have given. It may require more sophisticated and intelligent ways of finding the right grouping of elementary functions into processes. In the following chapters, we will elaborate and compare the methods of design space exploration during this design task.

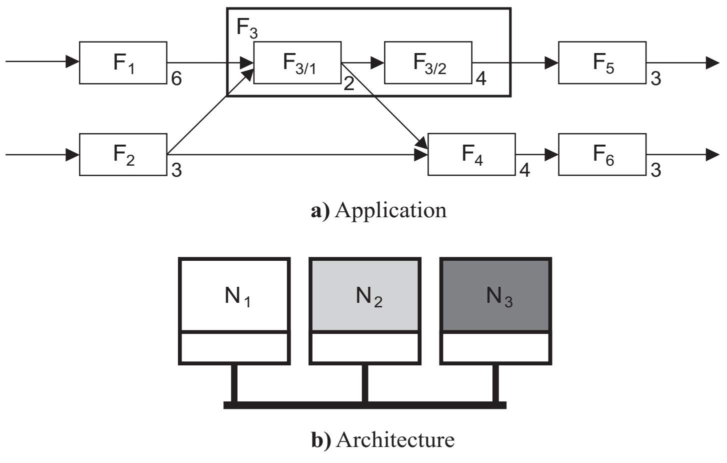
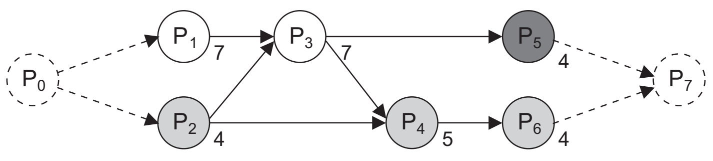
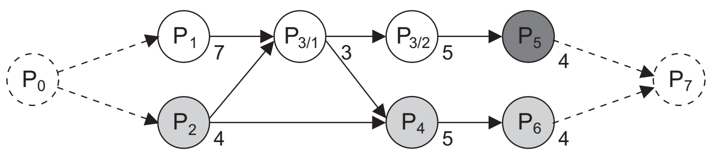


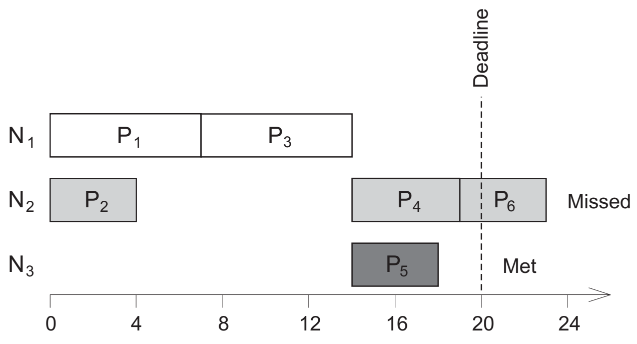
Fig. 3.5. Third motivational example (application and architecture)



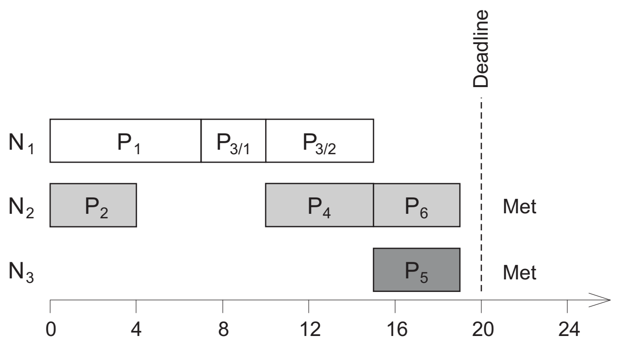
a) Process graph, case 1



b) Process graph, case 2



c) Scheduling, case 1



d) Scheduling, case 2

Fig. 3.6. Third motivational example (process graphs and scheduling)

4 Preliminaries: Time-Triggered Embedded Systems

In Chap. 2, we shortly described embedded real-time systems in general. In this chapter, we will dwell on a particular kind of embedded systems, time-triggered embedded systems, which are the focus of this thesis.

4.1 Time-Triggered versus Event-Triggered Systems

In a real-time computer system, every action, e.g., the execution of a process or the transmission of a message, must be initiated by a control signal. Such a control signal is called a *trigger*. Depending on the triggering mechanism for the start of processing and communication activities, two different approaches to the design of real-time applications exist, resulting in two types of systems [8]:

- Event-triggered approach (systems).
- Time-triggered approach (systems).

In *event-triggered* (ET) systems, all processing and communication activities are initiated by events. An *event* is an occurrence of a significant change of the system's state. For example, pressing of a button by the operator, an arrival of a message at a node, or termination of a process in a node. The signalling of events is realized by the interrupt mechanism. The worst-case execution time of an interrupt handler, as well as the context switches before and after the interrupt handler, are added up to the worst-case administrative overhead (WCAO) of the operating system. The fact that WCAOs due to interrupts are variable and not every interrupt leads to activating a different application process, makes the timing behaviour of ET systems difficult to predict. Event-triggered systems require an on-line scheduling strategy (e.g. preemptive fixed priority scheduling) in order to activate the appropriate process to service the event.

In *time-triggered* (TT) systems, all processing and communication activities are initiated by progression of time. The only interrupt available here is a periodic clock interrupt, which is generated whenever the real-time clock within a node reaches a preset value. At these instances, a so-called *trigger process* is invoked that scans the system for a change of state and decides whether another application process needs to be started. The scans introduce additional overhead. However, predictability of the WCAO in a TT system is high. Time-triggered systems are usually implemented with off-line scheduling, such as non-preemptive static cyclic scheduling. In a distributed TT system, it is assumed that the clocks of all nodes are synchronized to form a global notion of time.

Choosing an approach, the event-triggered or the time-triggered, depends on the particularities of the application. The high predictability of time-triggered systems makes the TT approach suitable for implementing safety-critical real-time systems. For complex distributed applications, such as modern automotive applications, the mixed event-triggered time-triggered approach can be used.

4.2 The Time-Triggered Protocol

Communication between the nodes in a distributed system is done using a communication protocol. The protocol used in time-triggered systems is the Time-Triggered Protocol (TTP). It was developed in the Vienna University of Technology and primarily intended for the automotive industry. It is a protocol for distributed real-time applications that require predictability and reliability (e.g. x-by-wire systems in vehicles). The description of the TTP given in this section is based on [9] and [11].

In a distributed system, nodes communicate over a broadcast channel. The Time-Triggered Protocol relies on the Time-Division Multiple Access (TDMA) scheme to control access to the channel (see an example in Fig. 4.1). Each node N_i is allowed to transmit only during a predetermined (off-line) time interval S_i, called *TDMA slot*. A node can send several messages packaged in one frame in its slot. If a node has no data to send, an empty frame is transmitted. A sequence of slots for every node in the system architecture forms a *TDMA round*. A node is allowed to transmit only once in a TDMA round. The duration of the slot of a given node and the sequence of all the slots is the same in every round; however, the amount and contents of data the node sends may vary from round to round. The sequence of all different TDMA rounds can be combined together in a cycle, called a *cluster cycle*, which is repeated periodically.

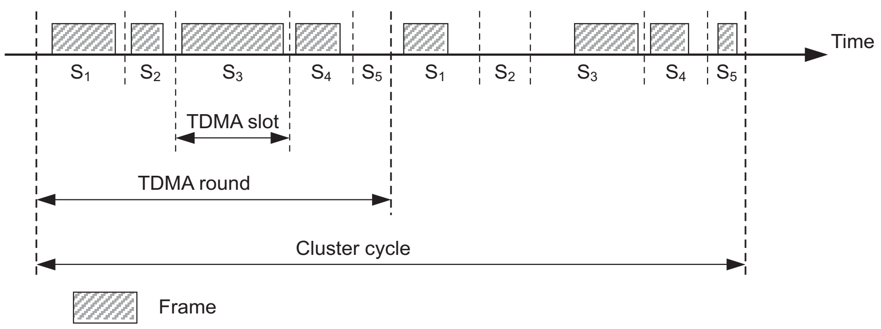


Fig. 4.1. TTP bus access scheme

The protocol services in a node are executed by its communication controller (see Fig. 2.2). Communication with the CPU is performed through the communication network interface. In case of TTP, the CNI is called *message base interface* (MBI), which is usually implemented as a dual ported RAM.

The DTMA bus access scheme is imposed by a so-called *message descriptor list* (MEDL), which is located in every communication controller (TTP controller). The MEDL determines when the TTP controller has to send a frame to or receive a frame from the communication channel. Thus, it serves as a schedule table for the controller. In addition, for each frame, the MEDL stores its address in the MBI, and its length. The size of the MEDL is one cluster cycle.

By executing a synchronization algorithm, the TTP controller provides synchronization of the local clock with the local clocks of all the other nodes.

4.3 The Hardware Architecture

In Sect. 2.2, we presented general hardware architecture for distributed real-time systems as a set of nodes interconnected with a communication network. The structure of a node is shown in Fig. 2.2. Fig. 4.2 gives a more detailed organization of a node in a time-triggered system. A typical node contains a CPU to execute the application, a real-time clock synchronized with the real-time clocks of all the other nodes, a RAM and a ROM to store the software, an input/output interface to sensors and actuators, a TTP controller to execute the time-triggered protocol, and it can also contain an ASIC to accelerate parts of the functionality.

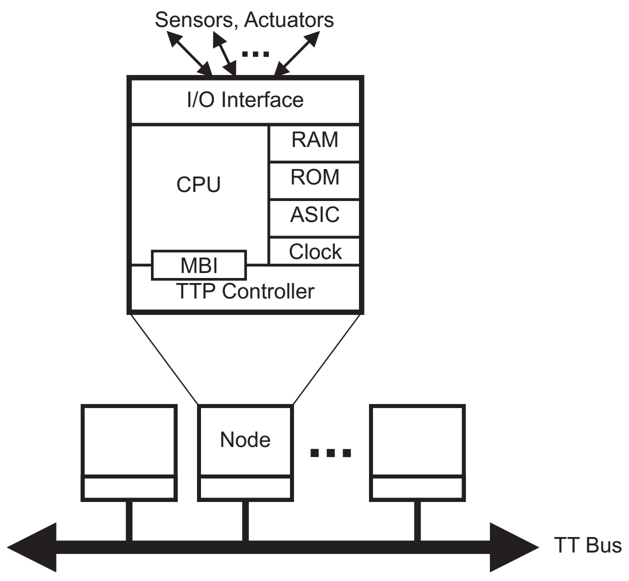


Fig. 4.2. The Hardware Architecture (adapted from [9])

4.4 The Software Architecture

The main component of the software architecture in a time-triggered system is a real-time kernel running in the CPU of each node. The kernel of a node has a schedule table that contains all the information needed to decide which process has to be executed or which message has to be transmitted at a given time instant in this node.

When a process finishes executing, it calls the send kernel function in order to send its messages. Sending a message between two processes mapped to the same node is done by copying its data from the memory location corresponding to the message of the first process to the memory location corresponding to the message of the second process. The time needed for this operation represents the WCAO for sending a message between processes located on the same node, δ_S. When the second process is activated, it will find the message in the right location. The scheduling policy is that whenever a receiving process needs a message, the message is already placed in the corresponding memory location. Thus, there is no overhead on the receiving side for messages exchanged within the same node [9].

In order to send a message to a process mapped to a different node, the kernel transfers the message to the TTP controller by packing it into a frame and placing it in the MBI. The worst-case administrative overhead of this operation is δ_{KS}. The TTP controller knows from its MEDL when it has to take the frame from the MBI and broadcast it on the bus. The TTP controller of the receiving node knows from its MEDL when it has to read a frame from the bus and transfer it to the MBI. The kernel reads a message from the MBI with the WCAO of δ_{KR}. When the receiving process is activated according to the local schedule table, it will already have the message in its memory location [9].

4.5 Static Cyclic Scheduling

Static cyclic scheduling is an off-line scheduling approach. The schedule built using this approach must guarantee all the deadlines, considering the precedence, resource, and synchronization requirements of the scheduled processes. In a distributed system, a schedule is constructed for each node, and plans both execution of the processes and access to the communication media. It will be used at run-time by a distributed scheduler.

A static cyclic schedule is a periodic time-triggered schedule. It contains activation times for all the processes executed by the node the schedule is built for. In this schedule, the time line is partitioned into a sequence of intervals called *minor cycles*. Each minor cycle's length, L_{minor}, is equal to the smallest of periods of the processes mapped to the current node,

$$L_{minor} = min(T_1, T_2, \dots, T_n). \quad (4.1)$$

The start of a minor cycle is denoted by a periodic clock interrupt. The whole schedule is called a *major cycle*. Its duration, L_{major}, is the least common multiplier of all the periods,

$$L_{major} = LCM(T_1, T_2, ..., T_n). \qquad (4.2)$$

The periods of the processes should be a multiple of the minor cycle time. If this is not the case, the periods must be adjusted. The number of minor cycles in a major cycle is determined as

$$N = L_{major} / L_{minor}. \qquad (4.3)$$

A process P_i is activated in every m-th minor cycle of a major one:

$$m = T_i / L_{minor}, \qquad (4.4)$$

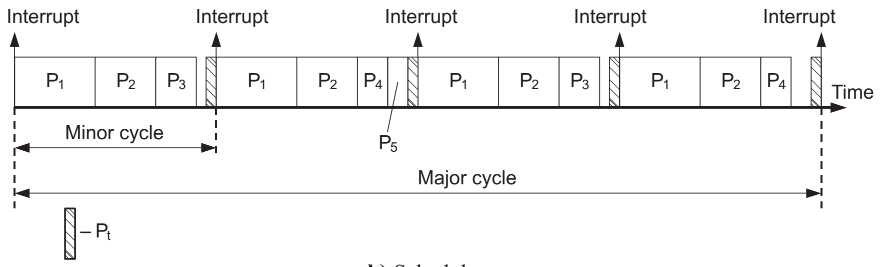
where T_i is the period of process P_i. The processes with execution times that do not fit into available time slots in the minor cycles should be split (off-line) into smaller parts. Then the schedule will be preemptive.

Example 4.1: Fig. 4.3 presents scheduling of five independent processes on one processor. In this example, $L_{minor} = 20$, $L_{major} = 80$. The trigger process should also be considered. Its period is made equal to the duration of the minor cycle. In the schedule chart (Fig. 4.3b), each rectangle denotes a process, and the width of a rectangle corresponds to the process's WCET.

In the case when all the processes have equal periods, there is only one minor cycle, which coincides with the major cycle.

Process	Period	WCET	
P_1	20	8	
P_2	20	6	
P_3	40	4	
P_4	40	3	
P_5	80	2	
P_t	20	1	P_t – trigger process

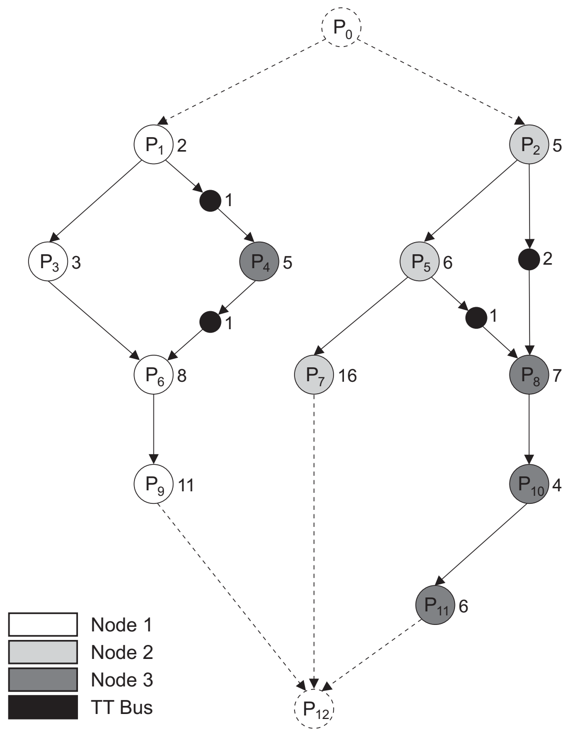
a) Process set



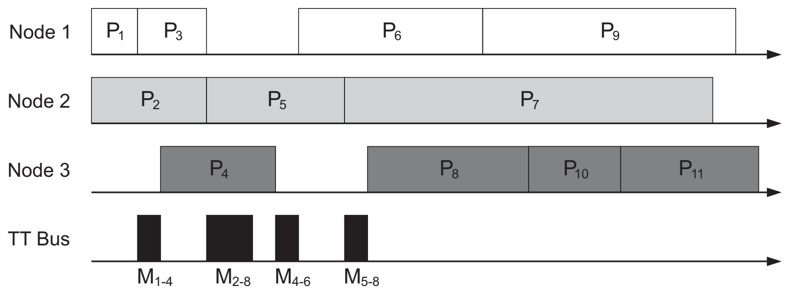
b) Schedule

Fig. 4.3. Example of scheduling independent processes with different periods

Example 4.2: Fig. 4.4 illustrates scheduling of a distributed application. The application is modelled as a process graph in which processes have equal periods (Fig. 4.4a). It has to be implemented on the architecture consisting of three nodes and a time-triggered bus. The chart (Fig. 4.4b) shows a possible schedule.



a) Application



b) Schedule

Fig. 4.4. Example of scheduling a distributed application

Optimal scheduling (e.g., minimizing the schedule length) has been proven to be an NP-complete problem. Heuristic approaches to the scheduling problem exist. They produce results of good quality within reasonable amount of time. A classical heuristic algorithm for static cyclic scheduling is the list scheduling algorithm (Fig. 4.5). The algorithm is based on maintaining a list of processes eligible for activation, *ReadyProcessesList*, for every processing element. Such a list contains the processes, mapped to the given processing element, that have not been scheduled yet, however, all their predecessors have been scheduled and terminated. The process to be scheduled next is the process with the highest priority in the *ReadyList*. Priorities are assigned to processes according to a priority function. An example of a priority function is a partial critical path from the given process to the sink process; the longer is the critical path the higher is the priority. A process is scheduled on that processing element which is available at the current time.

```
assign priorities to all the processes;
for each processing element PE do
  $IsAvailableTime_{PE} = 0$;
end for;
$CurrentTime = 0$;
repeat
  for each processing element PE do
    if $CurrentTime \geq IsAvailableTime_{PE}$ then
      update $ReadyProcessesList_{PE}$;
      select the highest priority process P from
             $ReadyProcessesList_{PE}$;
      schedule P at CurrentTime;
      $IsAvailableTime_{PE} = CurrentTime + WCET_{p}$;
    end if;
  end for;
  CurrentTime = the earliest of the termination times of the
                scheduled but not yet finished processes;
until all the processes are scheduled;
```

Fig. 4.5. List scheduling algorithm

5 Functional Blocks to Process Graph Translation

In Chap. 3, we described the problem of transitioning from a hierarchical behavioural application model to a flat behavioural application model represented as a graph of functional blocks and a process graph respectively. In this chapter, we propose strategies that can be used for such transitions.

5.1 Inputs and Outputs

According to the problem statement (see Sect. 3.2) the inputs to the problem are:

1. A graph of functional blocks,
2. System architecture, and
3. The synthesis algorithm.

The graph of functional blocks represents the system's functionality specified as a set of interacting hierarchical behaviours. The mathematical model of the graph was described in Sect. 3.1.1. We assume that the graph of functional blocks is pre-processed in such a way that:

- Global variables are removed; the functions have only local variables, and sharing of data is done by message passing,
- Loops are removed,
- All the functions have the same period of invocation,
- Dependences represent synchronous communication,
- The worst-case execution time (WCET) for each elementary function, and the worst-case transmission time (WCTT) for each dependence are given.

System architecture is specified as a set of computational nodes and a bus. The system is time-triggered. For each node, its performance is known and is given as a factor by which it differs from the reference node's performance. We consider the fastest node in the architecture as a reference. In the time-triggered bus, there is a TDMA slot for every node. We assume that all the slots are of the same size, and the size is given as the maximum of WCTTs of dependences between functions in the graph of functional blocks.

The synthesis algorithm is an algorithm that takes a process graph and system architecture as an input, allocates the processes from the process graph to the nodes of the system architecture[1], and produces schedule tables for every node using not-preemptive static cyclic scheduling approach. The implementation of the algorithm we will use is based on the balancing and several heuristic strategies (greedy, simulated annealing, tabu search) for mapping, and on the list algorithm for scheduling (see Sect. 4.5). Balanced mapping tries to distribute the computational load on the nodes evenly by simply choosing the least loaded node to map the current process to. Heuristic mapping tries to find the mapping that minimizes the schedule length. The synthesis routine takes into consideration the overhead of the trigger process. This overhead is expressed as a fraction of CPU power it utilizes, and, therefore, affects the execution times of the processes executed by the given CPU [9].

As a result of transformations applied to the graph of functional blocks, we want to get:

1. A process graph with minimal schedule length,
2. Mapping information for each node, that is which processes will be executed by the given node, and
3. Schedule tables for each node, containing the times at which the processes mapped to the given node are activated and the messages are transmitted.

5.2 Allocation Groups

The translation strategies we propose are based on the notion of an allocation group. An *allocation group* is a set of elementary functions of a graph of functional blocks that are allowed to be allocated to the same process. This means that functions belonging to a group may be executed within the same process or within different processes, however they may not be executed together with functions from other groups within the same process.

Having such allocation groups, translation of a graph of functional blocks to a process graph can be done using two approaches to allocation of these groups to processes, which we call *direct allocation* and *search-based allocation*. The translation strategies that use these approaches are, correspondingly, *straightforward* translations and *optimizing* translations. In the direct allocation, every group is directly allocated to a process. In the search-based allocation, these groups are partitioned into sub-groups in such a way that allocating each of the sub-groups to a process results in a schedule with the minimal schedule length. Hence, the task here is to find the optimal partitioning.

Allocation groups are identified according to grouping criteria, which can be the following:

- Compatibility of characteristics of elementary functions (e.g., the same period of activation),

[1] Since there is a single bus in our architecture, all the inter-node communications will be mapped to that bus.

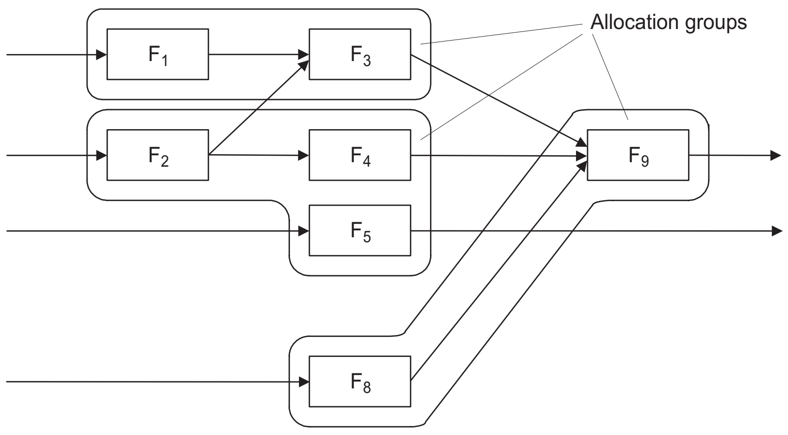


Fig. 5.1. Allocation groups based on constraints for the example graph in Fig.3.1

- Allocation constraints,
- The hierarchy of the graph of functional blocks, or
- No grouping.

Allocation constraints are design constraints on which elementary functions to allow to execute within the same process. Allocation groups can be formed based on them. For example, a design may have a requirement that particular functions must be executed on processing elements of a particular type (e.g. DSPs, or ASICs, etc.); therefore, these functions are included into one allocation group. Suppose that for the graph in Fig.3.1, we have the following constraints. Functions F_1 and F_3, functions F_2, F_4 and F_5, and functions F_8 and F_9 are allowed to be allocated to the same processes. Then, the allocation groups are as shown in Fig. 5.1. Such constraints may have different strictness. From saying that the functions in a group can be combined into processes in any way, to saying that the functions in a group must be a single process.

When the hierarchy of a graph of functional blocks is used as a grouping criterion, allocation groups are formed according to the containment of elementary functions in composite functions. Two cases of this kind of grouping are possible. In the first case, a group corresponding to each composite function at the highest level of hierarchy (topmost composite function) is created. For a given topmost composite function and the corresponding group, all the elementary functions contained in the composite function and all its sub-functions are included in the group. For example, for the graph of functional blocks in Fig.3.1, the allocation groups created based on topmost composite functions are shown in Fig. 5.2 In the second case, a group corresponding to every composite function at any level of hierarchy is created. For a given composite function and the corresponding group, all the elementary functions contained in the composite function are included in the group. For example, for the graph of functional blocks in Fig.3.1, the allocation groups created using this kind of grouping are shown in Fig. 5.3. In both cases, the elementary functions that do not belong to any composite functions are considered as separate groups. This criterion can be taken into consideration after the previous two in order to decrease the solution space if the search-based allocation is used.

A special type of grouping is no grouping. Alternatively, for the sake of commonality with the previous grouping types, we can say that a separate group for every elementary function is created. A given elementary function is included in the corresponding group.

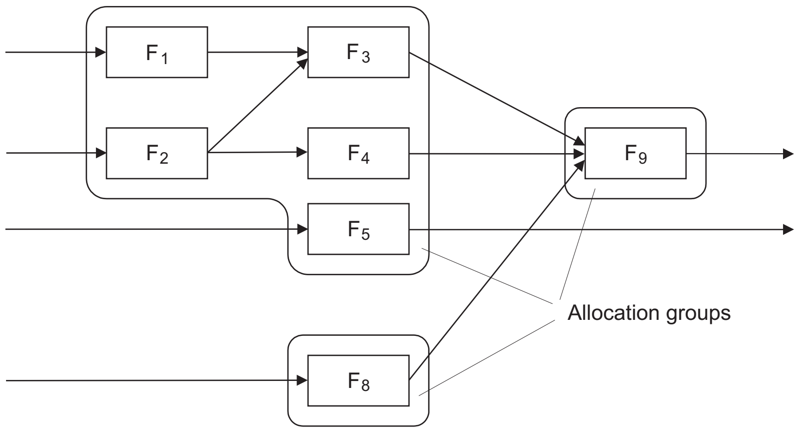


Fig. 5.2. Allocation groups based on topmost composite functions for the example graph in Fig.3.1

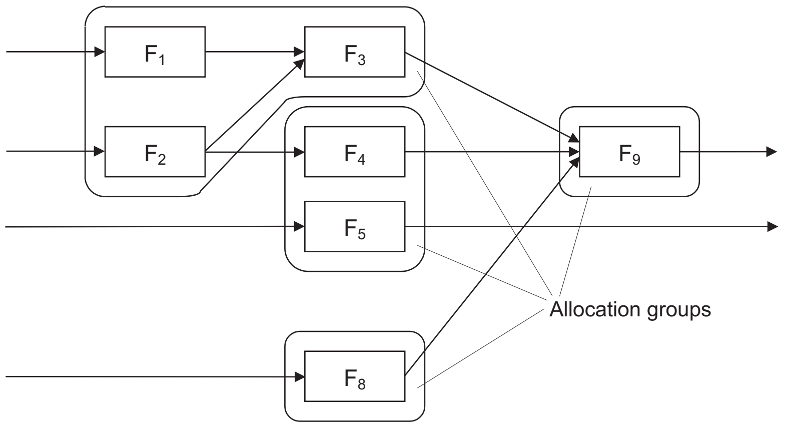


Fig. 5.3. Allocation groups based on composite functions for the example graph in Fig.3.1

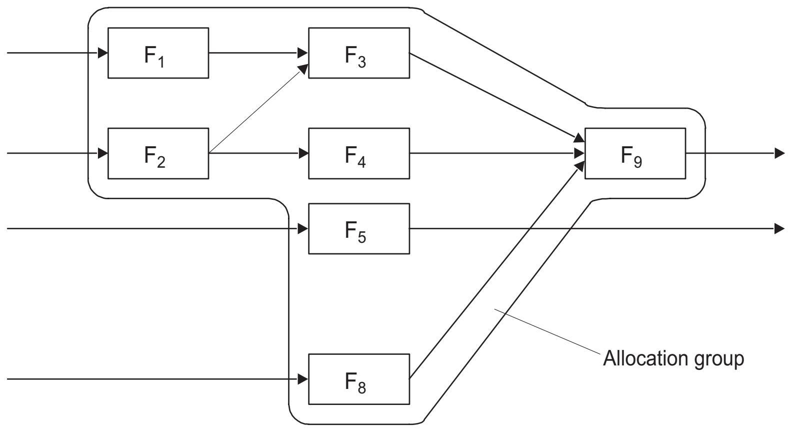


Fig. 5.4. An allocation group in the case with no constraints for the example graph in Fig.3.1

In the case when allocation is not constrained and no other grouping criteria is used, a single allocation group for the graph of functional blocks is created, which includes all the elementary functions (see the example in Fig. 5.4).

5.3 Worst-Case Execution Overhead, Execution Time and Transmission Time

As we have noted in the problem motivational examples in Chap. 3, when determining the worst-case execution time of a process, we should take into consideration the overhead introduced by the operating system when it executes the process. The sources making up this overhead when non-preemptive scheduling is used are the following:

- Process creation time,
- Process termination time, and
- Execution time of the system calls for message passing.

A process is created by the operating system via a create-process system call. During this operation, the system allocates resources to the process (e.g. memory), and creates the process control block (PCB) [10]. The time needed for this system call is the WCAO for creating a process, δ_C.

Termination of a process is done by executing an exit system call, which deallocates the resources and deletes the PCB [10]. The time this operation takes is the WCAO for terminating a process, δ_T.

The message passing mechanism in time-triggered systems was explained in Sect. 4.4. The worst-case administrative overheads associated with it are the WCAO for sending a message between processes located on the same node, δ_S, the WCAO for sending a message to a process located on a different node, δ_{KS}, and the WCAO for receiving a message from a process located on a different node, δ_{KR} [9].

The first two components of the overall process execution overhead, process creation time and process termination time, may or may not be present depending on when a process is created, during the initialization phase of the system or when it is time to execute it. In the first case, the process stays in the RAM all the time the system runs, and the scheduler activates it when needed. The process terminates only if the system shuts down. Activities that take place during initialization or shutdown are not captured in the schedule table; therefore, process creation and termination times are not considered as overheads. In the second case, the process is created every time it is scheduled for execution; hence, the associated overheads are taken into consideration.

In a time-triggered system, execution times of the above system calls are predictable, therefore, so is the process execution overhead.

The overall process execution overhead is different for every process. It depends on the length of the transmitted messages [9] and on the mapping. However, in this thesis, we assume it to be the same for all processes, and equal to 1 time unit.

Thus, the worst-case execution time of a process is composed of the WCETs of the elementary functions allocated to this process, and its worst-case execution overhead (WCEO):

$$C_i = \sum_{F_j \in F_{alloc.i}} C_{F_j} + \delta_{exec.i} \tag{5.1}$$

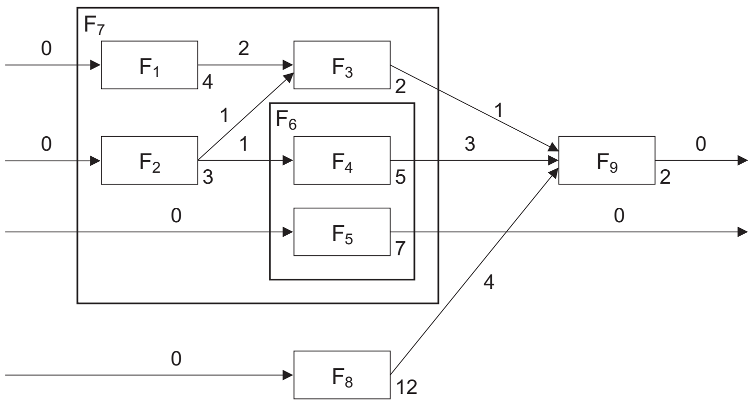
where C_i is the WCET of process P_i, C_{F_j} is the WCET of elementary function F_j, $F_{alloc.i}$ is the set of elementary functions allocated to process P_i, and $\delta_{exec.i}$ is the worst-case execution overhead for process P_i.

The worst-case transmission time of a message is the same as the WCTT of the corresponding dependence.

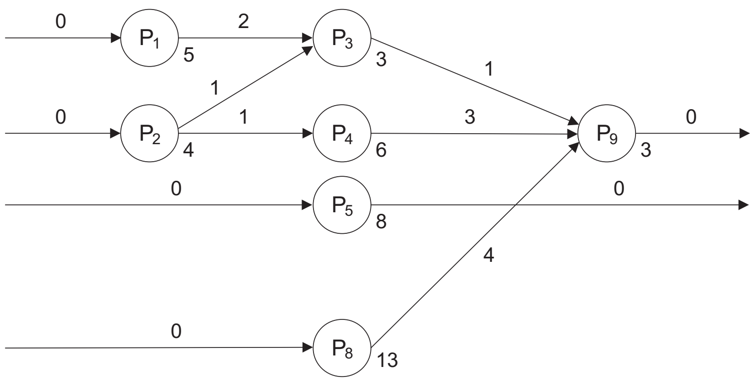
When performing transformations over a graph of functional blocks, there may be situations when several parallel dependences have to be merged into a single message. In this case, the WCTT of the message is determined as a sum of the WCTTs of the merged dependences (the amount of data to be transmitted is summed):

$$C_{i-k} = \sum_{D_{j-l} \in D_{alloc.i-k}} C_{D_{j-l}} , \tag{5.2}$$

where $C_{i\text{-}k}$ is the WCTT of message $M_{i\text{-}k}$ sent from process P_i to process P_k, $C_{D_{j-l}}$ is the WCTT of dependence $D_{j\text{-}l}$ between elementary functions F_j and F_l, $D_{alloc.i\text{-}k}$ is the set of dependences between the elementary functions allocated to process P_i and process P_k correspondingly.



a) Graph of functional blocks



b) Process graph

Fig. 5.5. Example of the translation based on direct allocation without grouping

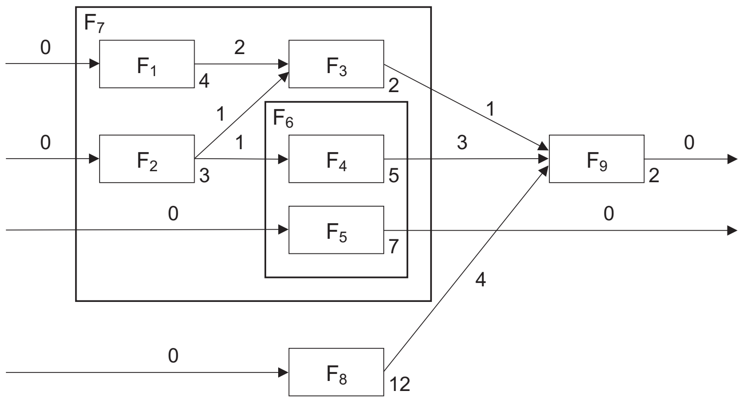
5.4 Translation Strategies

5.4.1 Straightforward Translation Strategies

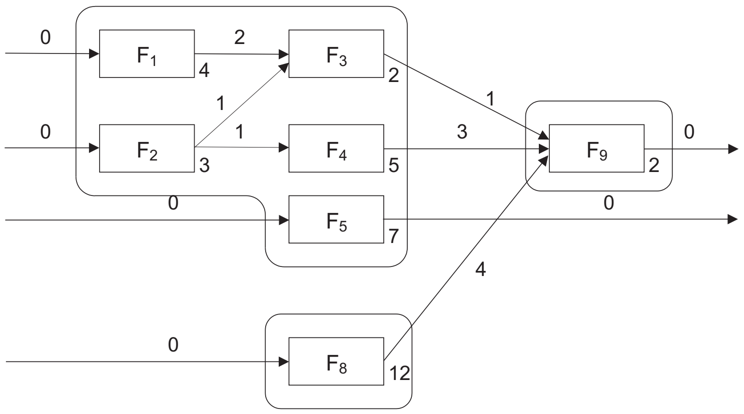
Straightforward translation strategies directly allocate an allocation group to a process. Let us consider two examples.

Example 5.1: Fig. 5.5 illustrates the translation based on direct allocation without grouping. The given graph of functional blocks with WCETs and WCTTs of elementary functions and dependences is shown in Fig. 5.5a, and the derived process graph – in Fig. 5.5b (the source and the sink processes are not shown).

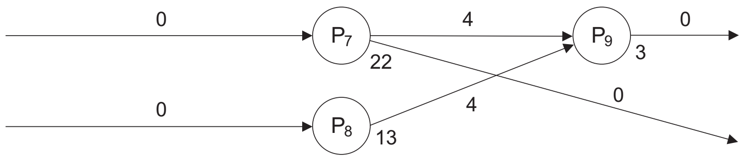
The translation is done by simply ignoring all the hierarchy in the graph of functional blocks and adding the process execution overhead to the WCET of each elementary function. The dependences between the elementary functions now become messages between processes with the same WCTTs.



a) Graph of functional blocks



b) Allocation groups



c) Process graph

Fig. 5.6. Example of the translation based on direct allocation with grouping according to topmost composite functions

Example 5.2: Consider the same graph of functional blocks as in the previous example. Then, using the translation based on direct allocation with grouping according to topmost composite functions, a process graph can be derived as shown in Fig. 5.6b, c. The worst-case execution and transmission times are calculated according to Eq. 5.1 and Eq. 5.2. For example, $F_{alloc.7} = \{F_1, F_2, F_3, F_4, F_5\}$, $F_{alloc.9} = \{F_9\}$, $D_{alloc.7\text{-}9} = \{D_{3\text{-}9}, D_{4\text{-}9}\}$, then $C_7 = C_{F_1} + C_{F_2} + C_{F_3} + C_{F_4} + C_{F_5} + \delta_{exec.7} = 4+3+2+5+7+1 = 22$,

$C_9 = C_{F_9} + \delta_{exec.9} = 2+1 = 3$, $C_{7-9} = C_{D_{3-9}} + C_{D_{4-9}} = 1+3 = 4$.

The straightforward translations do not search for a better solution in terms of schedule length. However, they are fast, and we can predict that in case of direct allocation without grouping very good scheduling results may be achieved if the process execution overheads are small, and an effective mapping optimization heuristic is used.

5.4.2 Optimizing Translation Strategies

The advantage of the straightforward translation strategies is that their execution is fast. However, the scheduling results they produce may be not satisfactory. In this subsection, we introduce *optimizing translation* strategies that use *search-based allocation* approach. These strategies try to find optimal allocation of elementary functions to processes.

Given a graph of functional blocks with defined allocation groups (an *allocation graph*), the task is to find the partitioning of these groups into sub-groups such that after allocating the sub-groups to processes a schedule of optimal (minimal) length is produced.

The best strategy, in terms of scheduling results, is to find all possible partitions and to choose the optimal one.

Example 5.3: Consider the graph with allocation groups shown in Fig. 5.3. Initial grouping is: $\{F_1, F_2, F_3\}$, $\{F_4, F_5\}$, $\{F_8\}$, $\{F_9\}$. Then, the possible partitions are the following:

1. $\{F_1, F_2, F_3\}$, $\{F_4, F_5\}$, $\{F_8\}$, $\{F_9\}$;
2. $\{F_1, F_2, F_3\}$, $\{F_4\}$, $\{F_5\}$, $\{F_8\}$, $\{F_9\}$;
3. $\{F_1, F_2\}$, $\{F_3\}$, $\{F_4, F_5\}$, $\{F_8\}$, $\{F_9\}$;
4. $\{F_1, F_2\}$, $\{F_3\}$, $\{F_4\}$, $\{F_5\}$, $\{F_8\}$, $\{F_9\}$;
5. $\{F_1, F_3\}$, $\{F_2\}$, $\{F_4, F_5\}$, $\{F_8\}$, $\{F_9\}$;
6. $\{F_1, F_3\}$, $\{F_2\}$, $\{F_4\}$, $\{F_5\}$, $\{F_8\}$, $\{F_9\}$;
7. $\{F_1\}$, $\{F_2, F_3\}$, $\{F_4, F_5\}$, $\{F_8\}$, $\{F_9\}$;
8. $\{F_1\}$, $\{F_2, F_3\}$, $\{F_4\}$, $\{F_5\}$, $\{F_8\}$, $\{F_9\}$;
9. $\{F_1\}$, $\{F_2\}$, $\{F_3\}$, $\{F_4, F_5\}$, $\{F_8\}$, $\{F_9\}$;
10. $\{F_1\}$, $\{F_2\}$, $\{F_3\}$, $\{F_4\}$, $\{F_5\}$, $\{F_8\}$, $\{F_9\}$.

In total, there are ten partitions of the given grouping, and a process graph corresponds to each of them. Each sub-group in a partition is allocated to a process. Next, every process graph is mapped and scheduled by the synthesis algorithm, and the one that gives the shortest schedule length is chosen as the solution.

An allocation group is a set (of elementary functions). The number of ways a set can be partitioned into disjoint subsets exponentially depends on the number of elements in the set, and is given by the so-called Bell number. For example, for the number of elements being *1*, *2*, ..., *14*, the Bell numbers have the values *1*, *2*, *5*, *15*, *52*, *203*, *877*, *4140*, *21147*, *115975*, *678570*, *4213597*, *27644437*, *190899322* [1]. For more information on set partitions, please refer to appendix A. For a grouping consisting of several allocation groups, the number of possible partitions is given by the following equation:

$$N_p = \prod_{i=1}^{N_A} b_i \ , \tag{5.3}$$

where b_i is the Bell number for the i-th allocation group, and N_A is the number of allocation groups.

With large allocation groups (consisting of many elementary functions), it may be not feasible to find all possible partitions of the given grouping, and to perform mapping and scheduling of corresponding process graphs within the acceptable time period. The solutions are:

- To decrease the number of possible partitions by making the allocation groups smaller and increasing their quantity. For example, let us suppose that there is an allocation group with *12* functions in it. This means that the group can be partitioned into sub-groups in *4,213,597* different ways. Making instead of this large group two smaller groups of size *7* and *5* would decrease the number of partitions to $877 \cdot 52 = 45{,}604$ (according to Eq. 5.3).
- To use a heuristic approach to searching for a partition that gives the optimal allocation. Any general-purpose heuristic algorithm can be used (e.g., neighbourhood search, simulated annealing, tabu search, genetic algorithms, etc.). The neighbourhood search, simulated annealing and tabu search algorithms are described in Appendix B.

In both exhaustive and heuristic approaches, the partitioning can be done using *partitioning vectors*. A partitioning vector for a graph of functional blocks with allocation groups is a collection of restricted growth strings for each allocation group in the graph. For instance, the partitioning vectors for the Example 5.2 are:

```
1. 000  00  0  0
2. 000  01  0  0
3. 001  00  0  0
4. 001  01  0  0
5. 010  00  0  0
6. 010  01  0  0
7. 011  00  0  0
8. 011  01  0  0
```

9. 012 00 0 0
10. 012 01 0 0

When heuristic algorithms are used, generation of neighbouring solutions of a current solution is necessary. A neighbour solution can be obtained by applying a simple design transformation to the current solution. For an allocation graph, a simple design transformation can be a move of a single elementary function to a new sub-group or to another existing sub-group, which is a partition of the initial allocation group the function belonged to. The neighbourhood of a given allocation graph is a set of other allocation graphs produced through such transformations. Applied to partitioning vectors, a move of an elementary function means a change of the value of the element corresponding to the function in the restricted growth string corresponding to the initial allocation group the function belonged to. For example, the neighbours of solution 011 00 0 0 are:

1. 001 00 0 0
2. 010 00 0 0
3. 012 00 0 0
4. 011 01 0 0

When using tabu search algorithm, a tabu history should be maintained. The history contains the partitioning vectors corresponding to the design transformations accepted during a number of the previous iterations. This is explained in the following example.

Example 5.4: The size of the history is 2. Suppose that the partitioning vector corresponding to the initial solution is 000 00 0 0. This vector is put into the history. During the next iteration the neighbour 000 01 0 0 was accepted. It is also pushed into the history. Vector 000 00 0 0 is a neighbour of the current vector but it is not considered because it is a tabu vector. Next, solution 010 01 0 0 was accepted. It is pushed into the history. However, since the size of the history is 2, the first vector in the history, which is 000 00 0 0, is popped out of it, and this means it is not prohibited to consider it as an alternative during the following iterations until it gets into the history again.

6 Experimental Results

Based on the theory presented in Chap. 5, six translation strategies were implemented. They are:

1. Direct allocation without grouping, or elementary function to process (EFP),
2. Direct allocation with grouping, or allocation group to process directly (AGPD),
3. Exhaustive search based allocation (AGPES),
4. Steepest descent neighbourhood search based allocation (AGPSD),
5. Simulated annealing based allocation (AGPSA), and
6. Tabu search based allocation (AGPTS).

The goal of the experiments is to compare the quality of the results produced by the six strategies and the time it takes to execute them for different sizes of the problem of translation.

6.1 Experimental Setup

The size of the problem depends on the number of allocation groups in a graph of functional blocks and on the number of elementary functions in an allocation group (see Eq. 5.3), and indirectly on the total number of elementary functions. For the experiments, we used the most difficult transformation case when there are no constraints on allocation of elementary functions to processes, which means that all the elementary functions are included in a single allocation group. Therefore, in this case, the size of the problem directly depends on the total number of elementary functions in a graph of functional blocks. In addition, for such a case, using the AGPD type of transformation does not make much sense; thus, it was not executed.

We considered problem dimensions of 10, 25, 50, 75, and 100 elementary functions, with hardware architectures consisting of 2, 2, 3, 4, and 4 nodes respectively. Ten graphs were randomly generated for each of the dimensions. In total, we evaluated 50 applications. The worst-case execution and transmission times for elementary functions and dependences were assigned randomly using uniform distribution.

The mapping algorithm used within the synthesis algorithm is the one of balanced mapping.

The experiments were run on a computer system with AMD Athlon 1.84 GHz processor, 1 GB of RAM, and MS Windows XP operating system.

6.2 Experimental Decisions

There is a number of decisions that had to be taken in order to achieve better results when using heuristic-based translation strategies.

1. Selection of the initial solution. Two alternatives exist here, either the solution produced by the EFP transformation or by the AGPD transformation. We tried both for the problem dimensions of 10 and 25 elementary functions. The first alternative showed better results. In addition to that, the heuristic-based translation with the second alternative for the initial solution may not always find a problem solution at all if the initial process graph and all others explored are cyclic or have messages with WCTTs that does not fit into corresponding slots in the TT bus. While the first alternative guarantees a solution which, in the worst case, is the initial one.
2. The cost function. As the cost function, we used the schedule length produced by the synthesis algorithm.
3. The cooling schedule and its parameters for the simulated annealing based translation. We used the cooling schedule by Lundy and Mees (eq. B.2). The initial temperature was selected such that any non-improving solution was accepted at temperatures close to the initial one. The low limit for the temperature parameter was set such that no non-improving solutions were accepted at temperatures close to the final one. In this way, the search is allowed to converge to a local optimum at its concluding stage. With β parameter, we controlled rigorousness of the search. We used the same values of the initial and final temperatures, which are 1000 and 10 respectively, for all sizes of applications and varied only β to achieve desired results.
4. The stopping condition and the tabu tenure (the size of the history record) for the tabu search based translation. The search is terminated after a given number of iterations without improvement has passed. By changing this parameter, we made the search more or less rigorous. [5] reports that the best results with tabu search heuristic for some size-dependent problems are achieved when the tabu tenure is in the range $[0.5\sqrt{N}, 2\sqrt{N}]$, where N is the size of the problem. For the dimensions we use, the respective integer ranges are [1, 7], [2, 10], [3, 15], [4, 18], [5, 20]. We ran the tabu search based translation for every value of tabu tenure in the ranges corresponding to dimensions of 10, 25, and 50 elementary functions. The best results were achieved for those tenures that are in the middle of a range. For example, for the size of 10, the value of tenure is 4, for the size of 25, the value is 6, and for the size of 50, the value is 9. Based on these results, we set the values of tenures for the sizes of 75 and 100 to 11 and 13 respectively.

Table 6.1. Parameters of the simulated annealing and tabu search algorithms

Problem size	Initial temperature	Final temperature	β	Tabu tenure	Iter. w/o improv.
10	1000	10	0.000009	4	100
25	1000	10	0.00001/0.000001	6	15/1000
50	1000	10	0.00003/0.000003	9	15/60
75	1000	10	0.00001	11	10/15
100	1000	10	0.00001, 0.000009, 0.000008	13	5/10

The values of β and tabu tenure are given for the normal and longer runs (normal run/longer run). For the size of 100, the simulated annealing based transformation was executed with 3 different values of β and the overall best results were chosen.

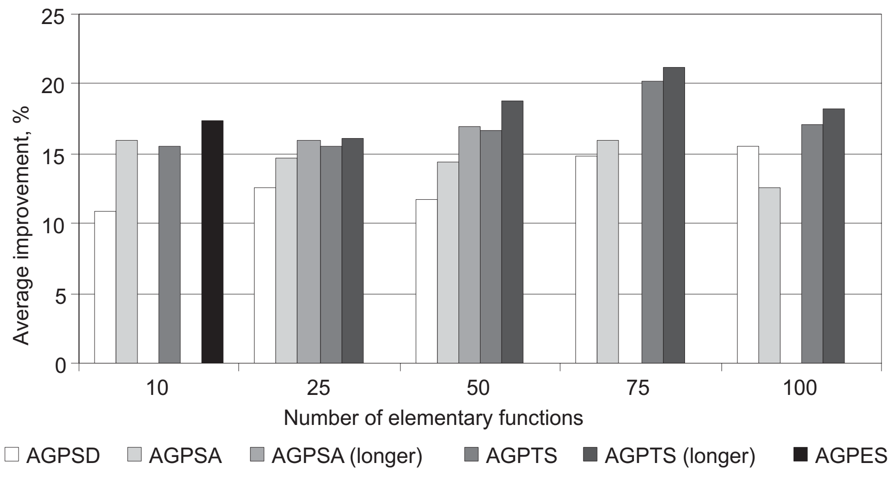
The experimentation strategy was the following. For function graphs of 10 elementary functions, all types of translations included in the setup were run. The necessary parameters of the simulated annealing and tabu search algorithms were adjusted to achieve results close to the ones produces by the exhaustive search algorithm. For function graphs of larger dimensions, execution of the exhaustive search based translation would take infeasible amount of time. Therefore, the parameters of the simulated annealing and tabu search algorithms were chosen such that to get results better than the ones produced by the neighbourhood search algorithm. Then, we showed that with longer runs it is possible to achieve even better results. The selected parameters are given in Table 6.1.

6.3 Comparison of Translation Strategies

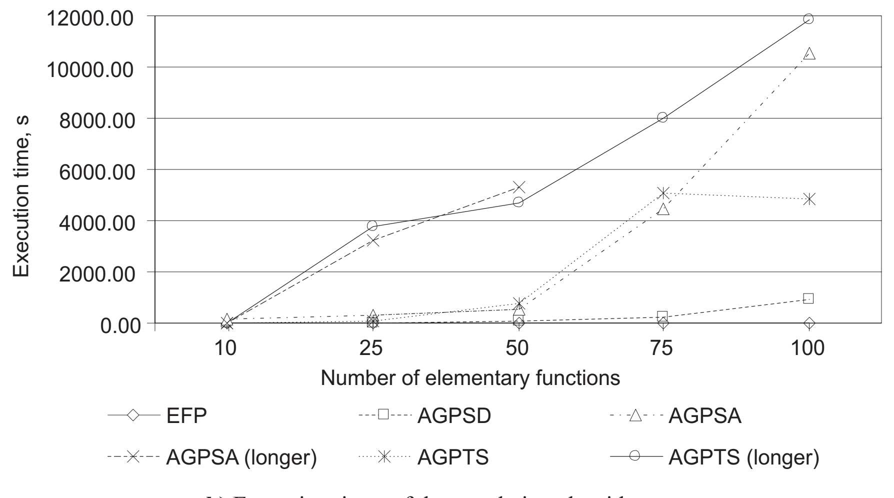
The chart in Fig.6.1a presents average schedule length improvements produced by optimizing translation strategies with respect to EFP translation. The average execution times of the translation algorithms are depicted in Fig. 6.1b. Please refer to Appendix C for the values of average schedule length deviations and execution times of the translation algorithms that were used as a source for Fig 6.1.

Experimenting with the simulated annealing based translation on graphs of 100 elementary functions using different parameters, we did not manage to get an improvement compared to the steepest descent based translation. We modified the simulated annealing algorithm in such a way that it does not accept any non-improving solutions for a number of iterations (temperatures) since the last improvement. This modified algorithm produced much better results, both schedule lengths and execution times, for the size of 100. We achieved an average improving deviation from the descent algorithm of 2.07%, and average execution time of 42m 22s. However, we did not get improvement for all ten graphs with a single value of β. Therefore, we had to use different values of β, which are 0.00009, 0.00008, 0.00007, 0.00006 and 0.00005, and to choose the best one and the corresponding solution for a given graph as a result.

The conclusion is that the optimizing translation strategies produce much better scheduling results than the straightforward ones. By adjusting the parameters of those based on simulated annealing and tabu search algorithms, it is possible to achieve the desired quality of results and the desired optimization times. However, simulated annealing needs to be tuned more thoroughly.



a) Quality of the translation algorithms



b) Execution times of the translation algorithms

Fig. 6.1. Comparison of the translation algorithms

7 Related Work

[13] and [7] focus on model transformations during the design of embedded software. The process of design and implementation of embedded software is partitioned into behaviour design, software structure design, runtime system design, and code generation. In each of these design stages, a different specification model is used – behavioural model, structural model, runtime model, and programming model. And in each stage, the corresponding model is refined into a model for the next stage.

The special attention is to transformation of a structural software model to a runtime model. The structural model is defined as a graph in which vertices are software components, and edges are links representing data or control flow between the components. The runtime model is represented as a task graph. A task may consist of subtasks executed in a predefined order. Each task has a set of attributes associated with it – period, deadline, execution location, and scheduling parameters. The links between tasks represent communications with given costs. The attributes of tasks and links are derived from the software structural model, platform model, and system constraints. Given such structural and runtime models, the problem of transformation of a structural software model to a runtime model is to map the actions of the components in the structural model to tasks in the runtime model in such a way that the execution sequence of actions defined in the structural model to achieve functional objectives is preserved, and the timing and scheduling constraints of the system are met.

The proposed transformation method relies on the notion of *transaction*. A transaction is a sequence of actions of software components performed in the end-to-end processing of an input event. During a transformation, all such transactions are identified. Each action in a transaction is assigned a priority. The actions with the same priority are grouped in a task. The communications between tasks are derived based on the links between components. The tasks are mapped to the given hardware platform and schedulability analysis is performed. The mapping is done using the first-fit algorithm to minimize the required usage of both processors and network links. The priorities of the actions in a transaction must be such that the schedulability and timing requirements of the transaction are satisfied. Changing an action's priority changes the task within which the action is executed, and hence, changes that task's timing attributes. Priority assignment, mapping, and schedulability analysis is an iterative process based on the simulated annealing heuristic. The optimal solution must satisfy timing and schedulability constraints, and must have low runtime overheads for task and inter-task communications.

8 Conclusions and Future Work

8.1 Conclusions

In this thesis, we addressed and solved the problem of automatic transitioning from a hierarchical behavioural application model represented as a graph of functional blocks to a flat behavioural application model represented as a process graph. We proposed strategies that can be used for such transitions. In order to confirm that the chosen approach is correct as well as to test the strategies, a tool translating graphs of functional blocks into process graphs was implemented, and experimentation on a number of randomly generated applications was done. The tool integrated the new translation algorithms with existing mapping and scheduling algorithms. It allows to explore the design space on three levels:

1. Different process graphs for a given graph of functional blocks,
2. Different mappings of a given process graph to the system architecture, and
3. Different schedule tables for a given mapping.

Not only schedule length can be used as a cost function within the tool, but also, for example, utilization of computational resources, or power and energy consumption, etc.

8.2 Future Work

In order to achieve better quality of results and to reduce optimization time, a problem specific heuristic translation algorithm is necessary. Such an algorithm could address the following issues:

1. Using feedback from the synthesis algorithm to guide the search process by analyzing the current solution and taking decision what the next solution should be.
2. Ability to tell if putting given functions into the same process will result in a better solution without running mapping and scheduling algorithms.
3. Allowing to create several instances of a function. Different instances will be included into different processes. This may improve the utilization of processing elements and save time on message passing.

More accurate modelling of process execution overheads is also needed.

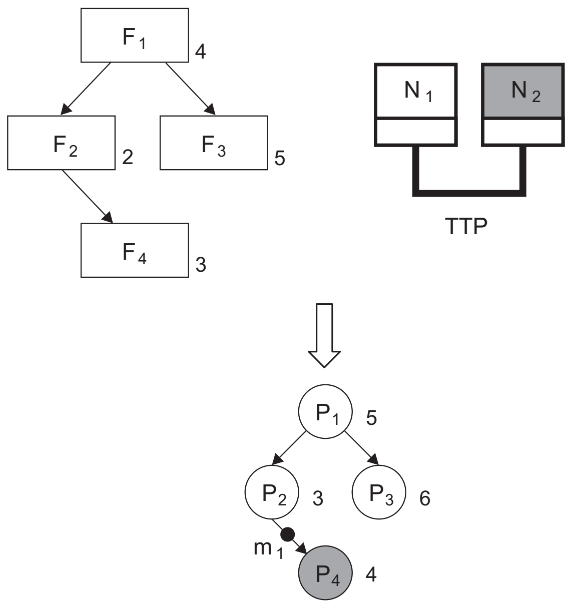
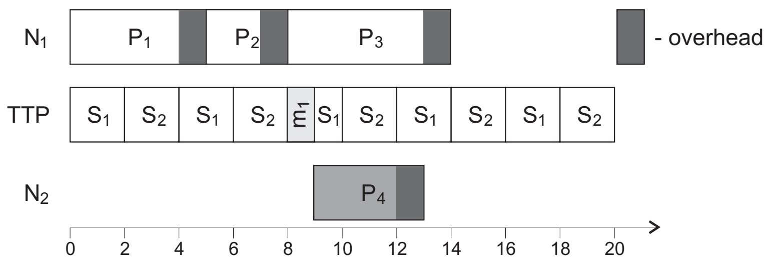


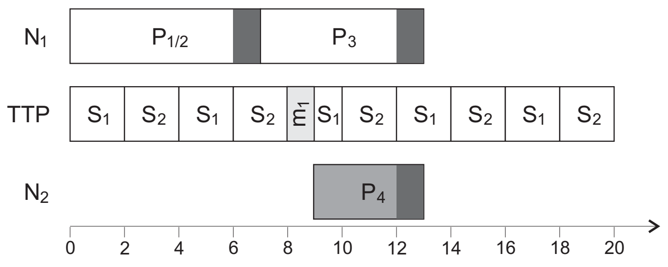
Fig. 8.1. Example application

The solution space to be explored may be reduced if the optimization is performed not on a graph of functional blocks but on a mapped and scheduled process graph. In this case, the simplest translation, when each elementary function is transformed into a process, can be used; and afterwards, a powerful mapping optimization algorithm, for instance, a tabu search based one. After we get the final mapped and scheduled model, further optimization can be done by merging some of the processes that have been mapped to the same node. Consider the following example.

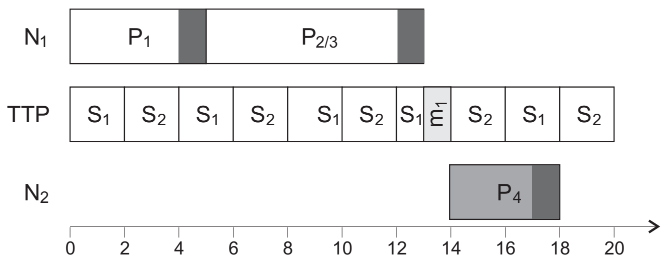
Example 8.1: Fig. 8.1 gives an application modelled as a graph of functional blocks, the system architecture, and the produced process graph with mapping, which we consider optimal. The numbers next to functions and processes are their worst-case execution times. Those of processes include the process execution overhead, δ_{exec}, which is equal to 1 time unit. The processes were scheduled as shown in Fig. 8.2a. We can see that three process, P_1, P_2, and P_3, were mapped to the same node and scheduled one after the other. These processes can be merged, and three alternatives exist, which are shown in Fig. 8.2b, c, d. Merging of n processes leads to reduction of the WCET of the resulting process compared to the total WCET of separate processes of $(n - 1)\delta_{exec}$. This may result in reduction of the schedule length as in alternative 1 (Fig. 8.2b). However, not necessarily, due to dependences between the merged processes and the processes mapped to other nodes, as in alternatives 2 and 3 in Fig. 8.2c,d.



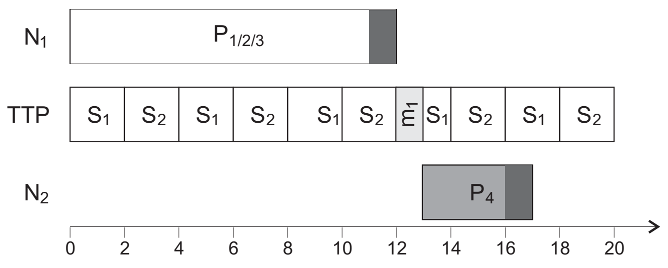
a) Initial schedule



b) Schedule after merging, alternative 1



c) Schedule after merging, alternative 2



d) Schedule after merging, alternative 3

Fig. 8.2. Schedules for the example application in Fig.8.1

Hypothetically, the rules of merging are:

- A process that transmits (P_i) can be merged with a process it depends on (P_j) only if P_j does not transmit, or P_i and P_j transmit to the same process, or P_i and P_j transmit to different nodes; and
- A process that transmits (P_i) can be merged with a concurrent process (P_k) only if P_i and P_k transmit to the same process, or P_i and P_k transmit to different nodes.

Otherwise, there will be no improvement in schedule length.

Appendix A

Set Partitions

The source of information provided here is [6]. A *set partition* of the set $[n]$ = $\{1, 2, \ldots, n\}$ is a collection B_0, B_1, ..., B_j of disjoint subsets of $[n]$ whose union is $[n]$. Each B_i is called a *block*. Below, the partitions for $n = 4$ are shown. The periods separate the individual sets so that, for example, 1.2.34 is the partition $\{\{1\}, \{2\}, \{3, 4\}\}$.

1 block: 1234
2 blocks: 123.4, 124.3, 12.34, 134.2, 13.24, 14.23, 1.234
3 blocks: 12.3.4, 13.2.4, 1.23.4, 14.2.3, 1.24.3, 1.2.34
4 blocks: 1.2.3.4

The blocks in each partition above are listed in increasing order of smallest element; thus block 0 contains element 1, block 1 contains the smallest element not in block 0, and so on. A *restricted growth string* (*RG string*) is a string $a_1a_2\ldots a_n$ where a_i is the block in which element i occurs. Here are the RG strings corresponding to the partitions shown above:

1 block: 0000
2 blocks: 0001, 0010, 0011, 0100, 0101, 0110, 0111
3 blocks: 0012, 0102, 0112, 0120, 0121, 0122
4 blocks: 0123

The name "restricted growth" comes from the fact that RG strings are characterized by the following growth inequality (for $i = 1, 2, ..., n-1$, and with $a_1 = 0$):

$$a_{i+1} \leq 1 + \max(a_1, a_2, ..., a_i\}. \tag{A.1}$$

The number of partitions of an n-set is called a *Bell number*, b_n. For $n = 0, 1, 2, ..., 14$, the Bell numbers have the values 1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, 678570, 4213597, 27644437, 190899322. The Bell numbers have the exponential generating function e^{e^x-1} and satisfy the recurrence relation

$$b_{n+1} = \sum_{k=0}^{n} b_k \binom{n}{k} \tag{A.2}$$

Appendix B

Heuristic Algorithms

B.1 Neighbourhood Search

Neighbourhood search overcomes the problem of computational expensiveness of searching the entire solution space for the optimal solution by searching only a small subset of the solution space. This is achieved by defining a neighbourhood structure on it and searching the neighbourhood of the current solution for an improvement. If there is no neighbour, which results in an improvement to the cost function, the current solution is taken as an approximation to the optimum. If an improvement is found, the current solution is replaced by the improvement and the process is repeated. The method of steepest descent searches the whole neighbourhood and selects that neighbour which results in the greatest improvement to the cost function. Random descent selects neighbouring solutions randomly and accepts the first solution, which improves the cost function. The algorithm of neighbourhood search with steepest descent is illustrated in Fig. B.1, where *N*(*s*) denotes the neighbourhood of *s*, which is a set of solutions reachable from *s* by a simple transformation; and *c*(*s*) is the cost of solution *s*.

```
Step 1     (Initialization)
  (A) Select a starting solution $s^{now} \in S$;
  (B) Record the current best known solution by setting
          $s^{best} = s^{now}$, $best\_cost = c(s^{best})$.

Step 2     (Choice and termination)
      Choose a solution $s^{next} \in N(s^{now})$ to satisfy $c(s^{next}) < c(s^{now})$
      and terminate if no such $s^{next}$ can be found.

Step 3     (Update)
      Reset $s^{now} = s^{next}$;
      **If** $c(s^{now}) < best\_cost$ **then** perform Step 1(B);
      **Goto** Step 2.
```

Fig. B.1. Algorithm of neighbourhood search with steepest descent

```
Select an initial solution $s^{now} \in S$;
Select an initial temperature $t_0 > 0$;
Select a temperature reduction function $\alpha$;
Repeat
    Repeat
        Randomly select $s^{next} \in N(s^{now})$;
        $\delta = c(s^{next}) - c(s^{now})$;
        If $\delta < 0$ then $s^{now} = s^{next}$ else
            generate random $x$ uniformly in the range (0, 1);
            If $x < \exp(-\delta / t)$ then $s^{now} = s^{next}$;
    Until $iteration\_count = nrep$
    Set $t = \alpha(t)$;
Until stopping condition = true.
Return $s^{now}$ as the solution.
```

Fig. B.2. Algorithm of simulated annealing

B.2 Simulated Annealing

The main disadvantage of neighbourhood search is its likelihood of finding a local, rather than global, optimum. By allowing uphill moves in a controlled manner, simulated annealing (SA) offers a way of alleviating this problem. The annealing algorithm is similar to the random descent method in that the neighbourhood is sampled at random. It differs in that a neighbour giving rise to an increase in the cost function may be accepted and this acceptance will depend on the control parameter called *temperature*, and the magnitude of the increase. The algorithm is given in Fig. B.2.

The simulated annealing algorithm has to be carefully tuned in order to provide good enough solutions in a short time. There is a number of generic and problem specific decisions that have to be taken to achieve this.

The generic decisions involve the *cooling schedule*, including the initial temperature (t_0), the number of iterations at a given temperature (*nrep*), the temperature reduction function (α), and the stopping criterion. The initial temperature must be "hot" enough to allow almost free exchange of neighbouring solutions. There are two most widely used temperature reduction schemes:

- Geometric reduction function

$$\alpha(t) = at, \tag{B.1}$$

  where $a < 1$. At each temperature a number of iterations is performed.
- Cooling schedule by Lundy and Mees

$$\alpha(t) = t / (1 + \beta t), \tag{B.2}$$

  where β is a suitably small value. Here, only one iteration is executed at each temperature. However, the temperature is reduced very slowly.

The stopping criteria can be the following: a small enough value of the temperature is reached, a number of iteration without acceptance has passed, the proportion of accepted and rejected solutions has dropped below a given value, or a given total number of iterations has been completed.

The problem specific decisions of simulated annealing deal with the solution space, neighbourhood structure, and the cost function.

B.3 Tabu Search

Tabu search (TS) is a neighbourhood search method that allows uphill moves in order to avoid local optima. However, compared to simulated annealing such moves are controlled in a more intelligent way. Tabu search maintains a selective history H of the states encountered during the search, and replaces $N(s^{now})$ by a modified neighbourhood $N(H, s^{now})$. History therefore determines which solutions may be reached by a move from the current solution, selecting s^{next} from $N(H, s^{now})$.

$N(H, s^{now})$ is a subset of $N(s^{now})$. $N(H, s^{now})$ is formed by excluding forbidden (tabu) neighbouring solutions from $N(s^{now})$. Tabus are used to restrict the search space and avoid cyclic behaviour. The algorithm of tabu search is presented in Fig. B.3.

```
Step 1 (Initialization)
   (A) Select a starting solution $s^{now} \in S$;
   (B) Record the current best known solution by setting
       $s^{best} = s^{now}$, $best\_cost = c(s^{best})$;
   (C) Set the history record $H$ empty.

Step 2 (Choice and termination)
       Determine $N(H, s^{now})$;
       Select $s^{next}$ from $N(H, s^{now})$ to minimize $c(s)$ over this set;
       Terminate by a chosen stopping condition and
       return $s^{best}$ as a result.

Step 3 (Update)
       Reset $s^{now} = s^{next}$;
       If $c(s^{now}) < best\_cost$ then perform Step 1(B);
```

Fig. B.3. Algorithm of tabu search

Appendix C

Evaluation of translation algorithms

Tables C.1 and C.2 present average schedule length deviations and average execution times produced by the translation algorithms. The deviations are given from EFP. In addition, for size 10, the deviations from AGPES are given for all the other algorithms.

Table C.1. Evaluation of the translation algorithms for the problem size of 10 elementary functions

Size	EFP		AGPSD			AGPSA			AGPTS			AGPES	
	Avg. dev. ES, %	Avg. exec. time	Avg. dev. EFP, %	Avg. dev. ES, %	Avg. exec. time	Avg. dev. EFP, %	Avg. dev. ES, %	Avg. exec. time	Avg. dev. EFP, %	Avg. dev. ES, %	Avg. exec. time	Avg. dev. EFP, %	Avg. exec. time
10	22.52	0.02s	-10.83	8.24	0.47s	-15.92	1.69	2m29.08s	-15.51	2.22	33.83s	-17.35	15m35.02s

A positive value of an average schedule length deviation means deterioration in schedule length

Table C.2. Evaluation of the translation algorithms for the problem size of 25, 50, 75 and 100 elementary functions

Size	EFP	AGPSD		AGPSA		AGPSA (longer)		AGPTS		AGPTS (longer)	
	Avg. exec. time	Avg. dev. EFP, %	Avg. exec. time	Avg. dev. EFP, %	Avg. exec. time	Avg. dev. EFP, %	Avg. exec. time	Avg. dev. EFP, %	Avg. exec. time	Avg. dev. EFP, %	Avg. exec. time
25	0.04s	-12.52	4.88s	-14.72	5m31.84s	-15.89	54m15.47s	-15.48	1m31.70s	-16.10	1h03m01.26s
50	0.17s	-11.69	42.85s	-14.43	8m51.60s	-16.88	1h28m38.75s	-16.7	12m58.16s	-18.79	1h18m24.45s
75	0.48s	-14.80	4m26.75s	-15.92	1h14m24.35s	–	–	-20.16	1h24m32.29s	-21.22	2h13m17.43s
100	1.02s	-15.56	15m09.92s	-12.60	2h56m02.49s	–	–	-17.12	1h20m46.33s	-18.19	3h18m02.01s

References

1. Bell number. http://mathworld.wolfram.com/BellNumber.html
2. Cortés LA, Eles P, Peng Z (1999) A survey on hardware/software co-design representation models. In: SAVE project report. Linköping University
3. Edgar SF (2002) Estimation of worst-case execution time using statistical analysis. Ph.D. thesis, University of York
4. Eles P (2003) Lecture notes on "System design and methodology: modelling and design of embedded Systems". http://www.ida.liu.se/~TDTS30/
5. Glover F, Laguna M (1993) Tabu search. In: Reeves CR (ed) Modern heuristic techniques for combinatorial problems. Blackwell scientific publications, Oxford
6. Info about set partitions. http://www.theory.csc.uvic.ca/~cos/inf/setp/SetPartitions.html
7. Kodase S, Wang S, Shin KG (2003) Transforming structural model to runtime model of embedded software with real-time constraints. In: Proceedings of Design, automation and test in Europe conference
8. Kopetz H (1997) Real-time systems: design principles for distributed embedded applications. Kluwer Academic Publishers, Boston Dordrecht London
9. Pop P (2003) Analysis and synthesis of communication intensive heterogeneous real-time systems. Ph.D. thesis, Linköping University
10. Silberschatz A, Galvin PB, Gagne G (2003) Operating system concepts. John Wiley & Sons, New York
11. The TTP Protocols. http://www.vmars.tuwien.ac.at/projects/ttp/
12. Turley J (1999) Embedded processors by the numbers. Embedded systems programming. Vol. 12 (5)
13. Wang S, Kodase S, Shin KG (2002) Automating embedded software construction and analysis with design models. In: Proceedings of Euro-uRapid conference