In *Evolutionary Algorithms for Embedded System Design*.
by Rolf Drechsler and Nicol Drechsler (Eds.).
In Genetic Algorithms and Evolutionary Computation (GENA).
© Kluwer Academic Publishers, Boston, Dordrecht, London, 2003.

Chapter 3

HIERARCHICAL SYNTHESIS OF EMBEDDED SYSTEMS USING EVOLUTIONARY ALGORITHMS

A Multi-Objective Approach

C. Haubelt, S. Mostaghim, F. Slomka, J. Teich, and A. Tyagi

Computer Engineering Laboratory (DATE)
Department of Electrical Engineering and Information Technology
University of Paderborn
D-33098 Paderborn, Germany
{haubelt, mostaghim, slomka, teich, tyagi}@date.upb.de

Abstract In this chapter, we propose an approach for the synthesis of heterogenous *embedded systems*, including allocation and binding problems. For solving these in general NP-complete problems, Evolutionary Algorithms have been proven to provide good solutions for search spaces of moderate size. For realistic embedded system applications, however, two more challenges must be considered: a) the complexity of the search space, and b) the multi-objective nature of the optimization problem to solve. I.e., the desired result of system synthesis is a design space exploration that provides the set of so-called Pareto-optimal solutions or an approximation thereof instead of just a single solution. Here, we propose a solution based on a *Multi-Objective Evolutionary Algorithm (MOEA)* which denotes a class of Evolutionary Algorithms that have recently proposed for *design space exploration* problems. Secondly, in order to reduce the complexity of typical search spaces, we propose a hierarchical problem and solution structure.

Keywords: Embedded System Design, Hierarchical Modeling, Multi-Objective Optimization, System Synthesis

1. Introduction

1.1 Motivation

New methodologies and tools are needed for the development of embedded systems as the market for such systems is steadily growing and the development cycles become shorter and shorter. Today, the spectrum of embedded systems covers devices for the control of all kinds of vehicles, the control of power plants, end user devices for telecommunication and devices for the infrastructure of telecommunication networks. One of the most popular embedded systems is the cellular telephone. In contrast to desktop computers, the functionality of an embedded system is restricted. Non-functional requirements like real-time constraints, data throughput and power consumption are important factors when designing an embedded system. Additionally, it is possible to implement different functions of the system on different hardware components. For example, a cellular telephone may be implemented using a single microprocessor. All functions will be performed in software on this processor. Such a system will be very expensive and the power consumption will be too high for a small portable device. On the other hand, a special hardware implementation is too inflexible to react to changes of requirements due to the dynamic market for cellular phones. The question which function to map to which hardware component on the discussed conditions is an optimization problem: This problem is called *hardware/software partitioning* or, more generally, *system synthesis* [3].

Previously, different kinds of processors such as signal processors and application specific hardware components had to be chosen by the designer. After the selection of components, the designer also had to define which function should be implemented on which of the selected components. If more then one function has to run on one hardware component, the schedule of the functions has to be defined, too. More formally, the resulting optimization problems turn out to be of combinatorial nature including *allocation* of components and assignment problems, i.e., *binding* functions to components and scheduling the functions according to multiple design constraints. Generally, the resulting search space of feasible allocation and binding solutions is a) complex and b) multi-objective. For example, speed, area, and power consumption are often similarly important, yet conflicting optimization goals.

1.2 Related Work

Due to the large complexity of the design space, heuristic optimization techniques are required to solve the optimization problem. Different heuristic optimization techniques are discussed in the literature for hardware/software partitioning or system synthesis. One of the first design tools for mixed hardware/software systems called COSYMA [9] is based on Simulated Annealing

[13]. Tabu-Search as an optimization technique for hardware/software partitioning is described in [8]. A more general approach for system-level synthesis can be found in [3] and [6]. In both papers, Genetic Algorithms are used to explore the complete set of optimal system architectures. While the focus of [3] is on data-dominated systems, the approach in [6] is targeted to periodic real-time systems.

A comparison of Simulated Annealing, Tabu-Search and Genetic Algorithms as approaches for system synthesis can be found in [2] and [7]. The synthesis approach in [2] is based on a new real-time response time analysis technique while in [7] only scheduling of tasks on a given transputer architecture is discussed and the latency of the application is minimized.

1.3 Contribution

In this chapter, we focus on two extensions to previous approaches to system synthesis: a) *hierarchical problem decomposition* and, based on this extension, b) *hierarchical design space exploration*. In previous approaches such as [3], [6], and others, both the application specification and the system architecture are modeled non-hierarchically. E.g., the application is modeled by a task graph with a relatively few number of coarse grain tasks. The hardware architecture is also modeled coarse grain, describing allocatable objects such as buses, processors and hardware co-processors.

Here, we introduce a new hierarchical approach to model embedded systems for synthesis which allows the specification of design alternatives of the application algorithm as well as alternatives of different hardware architectures. For example, the error correction of transmitted data of a mobile telephone can be performed by modern coding algorithms or in a more classical way by implementing complex protocol functions like data retransmission.

The modeling of alternatives at both the algorithm as the architecture level extends the search space as compared to fixed problems. The hierarchical variety leads to an explosion of the search space for feasible allocations and bindings. During a design space exploration, one is also not interested in just a single optimal solution, but in the set of *all* optimal solutions, the so-called *Pareto-set*.

Here, we therefore propose two means to deal with the increased complexity of the problem: *Pareto-Front Arithmetics* and a *Hierarchical Multi-Objective Evolutionary Algorithm*. The idea of Pareto-front arithmetics is to first explore all the Pareto-fronts resulting from mapping each leave node of a given hierarchical specification and later to combine these Pareto-fronts to generate an approximation of the Pareto-front on higher hierarchical levels iteratively. We will show that, depending on the conditions of feasible allocations and bind-

ings as well as on the objective functions to be optimized, the constructed front might not be the true Pareto-front, but an approximation thereof.

In the second approach, a Hierarchical Multi-Objective Evolutionary Algorithm is proposed, introducing a chromosome structure that reflects the hierarchical nature of the specification, and special techniques for avoiding infeasible solutions are presented to reduce the search space.

In both presented approaches, a special Evolutionary Algorithm adapted to multi-objective optimization problems is used, a so-called *MOEA* (*Multi-Objective Evolutionary Algorithm*). We capture the results of both approaches for the example of an MPEG4 coder which is the running example of this chapter. The objectives to be optimized are cost, power, and the *flexibility* of a design. The flexibility is defined here as the richness of different functions a system implements.

2. A Model for Embedded System Synthesis

This section introduces the required mathematical background, i.e., a graph-theoretic model is introduced in order to establish a formal definition of system synthesis.

2.1 The Specification Graph Model

Blickle et al. propose a graph-based approach for embedded system optimization and synthesis [3]. We introduce this model as a starting point and derive our enhanced hierarchical model subsequently based on this basic model.

The specification model [3] consists of two main components:

- A given functional specification that should be mapped onto a suitable architecture of hardware components as well as the class of possible architectures are described each by means of a universal directed *dependence graph* $g(V, E)$.
- The user-defined mapping constraints between tasks and architecture components are specified in a specification graph $g_\mathrm{s}(V_\mathrm{s}, E_\mathrm{s})$. Additional parameters which are used for formulating the objective functions and further functional constraints may be assigned to either vertices or edges of g_s.

At first, the (well known) concept of a dependence graph is used to describe the functional specification as well as the target architecture variety.

Definition 3.1 (Dependence Graph) *A* dependence graph *g is a directed graph $g(V, E)$. V is a finite set of vertices and $E \subseteq (V \times V)$ is a set of edges.*

For example, the dependence graph to model the data flow dependencies of a given specification will be termed *problem graph* $g_\mathrm{p} = (V_\mathrm{p}, E_\mathrm{p})$ in the follow-

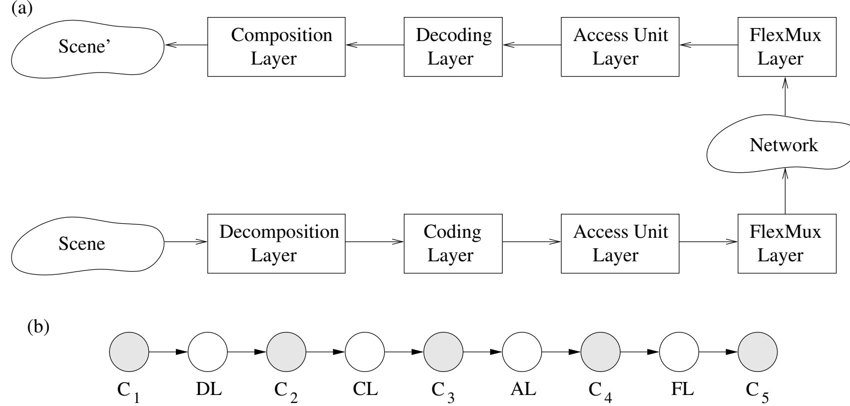

Figure 3.1. (a) Behavioral specification of an MPEG4 coder and (b) corresponding problem graph.

ing. Here, V_{p} is the set of vertices which model either functional operations or communication operations. The edges in E_{p} model dependence relations, i.e., define a partial order among the operations.

Example 3.1 The example introduced here and used throughout this chapter is an MPEG4 coder. Due to its complexity and its clear decomposition into subsystems, the MPEG4 coder seems to be a reasonable case study for our new approach. Its block diagram is shown in Figure 3.1(a). We start with a given scene, where the scene is either natural, synthesized or both. This scene is decomposed into audio/visual objects like images, video, animated 2D meshes, speech, synthesized sounds, etc. Each audio/visual object (AVO) is coded by an appropriate coding algorithm (indicated by the coding layer). In the next step (Access Unit Layer), the data are provided with time stamps, data type (audio, video), clock references, etc. The FlexMux Layer (Flexible Multiplexer) allows to group streams with the same QoS (quality of service) requirements. The upper part of Figure 3.1(a) shows the MPEG4 decoder which should be able to reconstruct the coded scene as good as possible. Since we are only interested in the MPEG4 coder, we omit the explanation of the decoder part.

The behavioral specification of the coder (lower part of Figure 3.1(a)) is refined to the problem graph shown in Figure 3.1(b). Between two data flow dependent operations, we insert an additional vertex in order to model the required communication.

The architecture including functional resources and buses can also be modeled by a dependence graph termed *architecture graph* $g_{\mathrm{a}} = (V_{\mathrm{a}}, E_{\mathrm{a}})$. V_{a} may consist of two subsets containing functional resources (hardware units like an

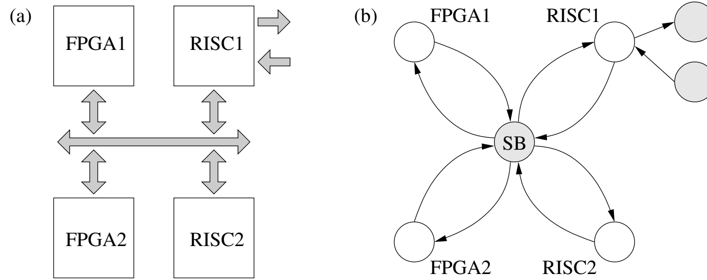

Figure 3.2. (a) Architecture template to implement the problem graph of Figure 3.1 and (b) corresponding architecture graph.

adder, a multiplier, a RISC processor, a dedicated processor, or an ASIC) and communication resources (resources that handle the communication like shared buses or point-to-point connections). An edge $e \in E_\mathrm{a}$ models a directed link between two resources. All the resources are viewed as *potentially allocatable* components.

Example 3.2 The problem graph given in Example 3.1 is mapped onto a target architecture shown in Figure 3.2(a). The architecture consists of four functional resources, two programmable RISC processors, two field programmable gate arrays (FPGAs), and a single shared bus (SB). Additionally, the processor RISC1 is equipped with two special ports. The corresponding architecture graph is given in Figure 3.2(b).

Next, it is shown how user-defined mapping constraints can be specified in a graph based model. Moreover, the *specification graph* will also be used to define *binding* and *allocation* formally.

Definition 3.2 (Specification Graph) *A* specification graph $g_\mathrm{s}(V_\mathrm{s}, E_\mathrm{s})$ *consisting of a problem graph* $g_\mathrm{p}(V_\mathrm{p}, E_\mathrm{p})$*, an architecture graph* $g_\mathrm{a}(V_\mathrm{a}, E_\mathrm{a})$*, and a set of* mapping edges E_m*. In particular,* $V_\mathrm{s} = V_\mathrm{p} \cup V_\mathrm{a}$*,* $E_\mathrm{s} = E_\mathrm{p} \cup E_\mathrm{a} \cup E_\mathrm{m}$*, where* $E_\mathrm{m} \subseteq V_\mathrm{p} \times V_\mathrm{a}$.

Consequently, mapping edges relate the vertices of the problem graph to vertices of the architecture graph. The edges represent user-defined mapping constraints in the form of a relation: “can be implemented by”.

Example 3.3 Figure 3.3 shows an example of a specification graph including the problem graph of Example 3.1 and the architecture graph of Example 3.2. The dashed edges between the two graphs are the additional mapping edges E_m that describe all possible mappings. For example, operation DL can be executed only on RISC1. Note that it can be useful to map communication

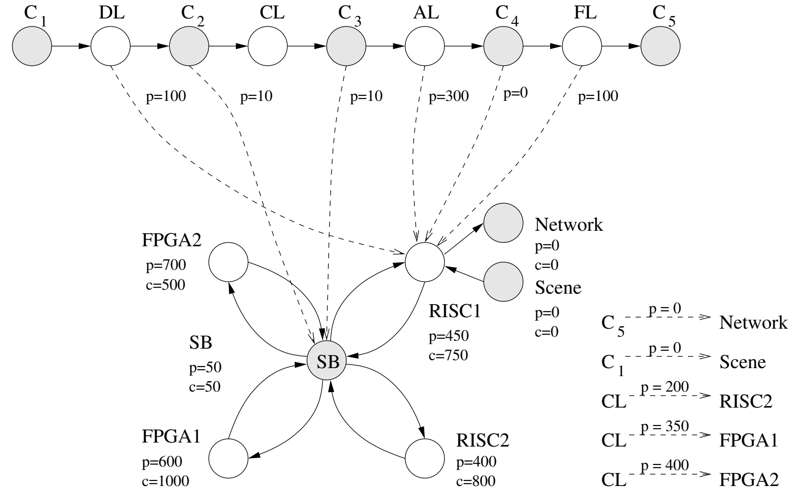

Figure 3.3. Specification graph for the MPEG4 coder.

nodes of the problem graph to functional resources: If both predecessor and successor node of a communication node are mapped to the same functional resource, no communication is necessary and the communication is *internal*. In this case, the communication can be viewed to be handled by the functional resource. For the purpose of better visibility, additional mapping edges are depicted in the lower right corner of Figure 3.3.

Only the coding layer (CL) can be executed on the alternative resources RISC2, FPGA1, and FPGA2. The mapping edges $e(v_{\mathrm{p}}, v_{\mathrm{a}}) \in E_{\mathrm{m}}$ are annotated with additional power consumptions which arise when operation v_{p} is executed on resource v_{a}. Furthermore, all resources in Figure 3.3 are annotated with allocation cost and power consumptions. These values have to be taken into account if the corresponding resource is used in an implementation of the problem.

In the above way, the model of a specification graph allows a flexible expression of the expert knowledge about useful architectures and mappings.

2.2 Hierarchical Modeling

By looking at the MPEG4 standard, one can see that the coding layer consists of several different coding schemes. Although the specification graph allows the modeling of alternative mappings, it cannot express the knowledge about possible refinements of a given task. Also, vertices of the architecture graph can be decomposed into several submodules, e.g., an FPGA can be configured with various designs or the coarse grain RISC processor represents a full class of processors.

In order to model these refinements, we propose a hierarchical extension to the previously introduced model of a specification graph. The following *hierarchical specification graph* is thus based on the (non-hierarchical) specification graph described in the previous section and the concept of *hierarchical graphs*.

Definition 3.3 (Hierarchical Graph) *A* hierarchical graph $g(V, E)$ *is a pair* (V, E)*, where*

- V *denotes the set of vertices as defined in Definition 3.4, and*
- $E \subseteq \left\{\bigcup_{v \in V} v.O\right\} \times \left\{\bigcup_{v \in V} v.I\right\}$ *denotes the set of edges, where* $v.O$ *and* $v.I$ *denote the set of outputs and inputs of vertex* $v \in V$*, respectively.*

As mentioned above, a vertex can be refined by a set of subgraphs.

Definition 3.4 (Hierarchical Vertex) *A* hierarchical vertex $v \in V$ *is a 4-tuple* (I, O, G, m)*, where*

- $I = \{i_1, i_2, \ldots, i_\mathrm{n}\}$ *is a finite set of inputs, also denoted* $v.I$*,*
- $O = \{o_1, o_2, \ldots, o_\mathrm{m}\}$ *is a finite set of outputs, also denoted* $v.O$*,*
- $G = \{g_1, g_2, \ldots, g_\mathrm{l}\}$ *is a finite set of subgraphs, also denoted* $v.G$*, and*
- $m : \{I_\mathrm{G} \cup O_\mathrm{G}\} \to \{I \cup O\} = \begin{cases} m(i) & \text{if } i \in I_\mathrm{G} \\ m(o) & \text{if } o \in O_\mathrm{G} \end{cases}$ *, where*
  - $m(i) : I_\mathrm{G} \to I$ *with* $I_\mathrm{G} = \left\{\bigcup_{g \in G} \bigcup_{\widetilde{v} \in g.V} \widetilde{v}.I\right\}$ *is a function which uniquely maps the inputs* $i \in \widetilde{v}.I$ *of each vertex* $\widetilde{v} \in g.V$ *of each subgraph* $g \in G$ *to an input* $i \in v.I$ *of* v*, and*
  - $m(o) : O_\mathrm{G} \to O$ *with* $O_\mathrm{G} = \left\{\bigcup_{g \in G} \bigcup_{\widetilde{v} \in g.V} \widetilde{v}.O\right\}$ *is a function which uniquely maps the outputs* $o \in \widetilde{v}.O$ *of each vertex* $\widetilde{v} \in g.V$ *of each subgraph* $g \in G$ *to an output* $o \in v.O$ *of* v*.*

In the following we use the term $P = I \cup O$ *to denote the set of ports* $p \in P$ *associated with the vertex* v*. Also, the terms* graph *and* vertex *are used for their hierarchical counterparts, respectively.*

If a set of subgraphs $G' \subseteq v.G$ is selected as a refinement of a vertex v, we are able to flatten our model, i.e., to replace the vertex v by the selected subgraphs $g \in G'$.

Example 3.4 Figure 3.4 shows possible refinements of the coding layer of the MPEG4 coder shown in Figure 3.1. There are two types of codings: audio

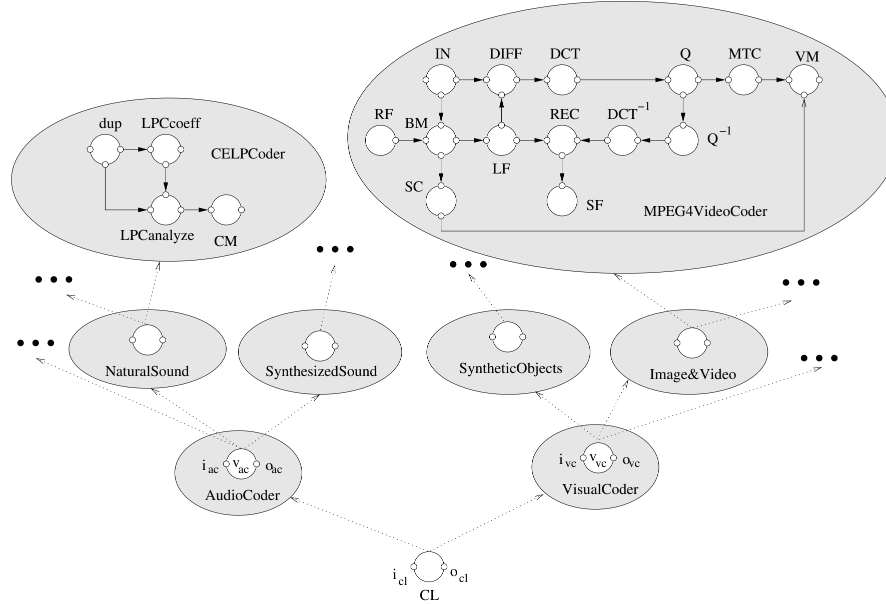

Figure 3.4. Refinements of the coding layer of the MPEG4 coder shown in Figure 3.1.

and visual coding, i.e., $\mathrm{CL}.G = \{\mathrm{AudioCoder}, \mathrm{VisualCoder}\}$. The input and output set of CL contains only one element ($\mathrm{CL}.I = \{\mathrm{i_{cl}}\}, \mathrm{CL}.O = \{\mathrm{o_{cl}}\}$).

The audio coder subgraph consists of a single vertex $\mathrm{AudioCoder}.V = \{\mathrm{v_{ac}}\}$ and no edges ($\mathrm{AudioCoder}.E = \{\}$). In the next level of the hierarchy, we can refine $\mathrm{v_{ac}}$ by $\mathrm{v_{ac}}.G = \{\mathrm{NaturalSound}, \mathrm{SynthesizedSound}, \ldots\}$. The set of inputs and outputs for vertex $\mathrm{v_{ac}}$ is given by $\mathrm{v_{ac}}.I = \{\mathrm{i_{ac}}\}$ and $\mathrm{v_{ac}}.O = \{\mathrm{o_{ac}}\}$.

One of the coding schemes for natural sounds is the CELP algorithm (Code Excited Linear Prediction) depicted in the upper left corner of Figure 3.4. An incoming speech signal is filtered by a Linear Prediction Coding (LPC) to produce a noise-like residual. After determining the filter coefficient on a frame of 80 speech samples, these samples are filtered one by one. Blocks of 40 filter outputs are compared with a codebook of 1024 reference blocks. All vertices in the CELP leaf subgraph are leaf vertices, since $\mathrm{dup}.G = \ldots = \mathrm{CM}.G = \{\}$.

The visual coding consisting of a single vertex $\mathrm{v_{vc}}$ can be divided into image and video coding, synthetic objects coding, compression of 2D and 3D meshes, ... The compression algorithm for images and videos is part of the MPEG4 standard and is depicted in the upper right corner of Figure 3.4. Motion estimation is done by the block matching operation (BM) and the block subtraction is named DIFF (difference). Other blocks in Figure 3.4 are: quantization (Q), shape coding (SC), motion texture coding (MTC), and video multiplex (VM).

Again, all vertices in the MPEG4 leaf subgraph are leaf vertices, i.e., the set of associated subgraphs is empty.

With the inputs ($\mathrm{v_{vc}}.I = \{\mathrm{i_{vc}}\}$) and outputs ($\mathrm{v_{vc}}.O = \{\mathrm{o_{vc}}\}$) of vertex $\mathrm{v_{vc}}$, we determine the port sets of the subgraphs associated with the coding layer vertex CL:

$$\begin{aligned} \mathrm{CL}.I_{\mathrm{G}} &= \{\mathrm{i_{ac}}, \mathrm{i_{vc}}\} \\ \mathrm{CL}.O_{\mathrm{G}} &= \{\mathrm{o_{ac}}, \mathrm{o_{vc}}\} \end{aligned}$$

The mapping function of the coding layer is $\mathrm{CL}.m(\mathrm{i_{ac}}) = \mathrm{i_{cl}}, \mathrm{CL}.m(\mathrm{i_{vc}}) = \mathrm{i_{cl}}, \mathrm{CL}.m(\mathrm{o_{ac}}) = \mathrm{o_{cl}}$ and $\mathrm{CL}.m(\mathrm{o_{vc}}) = \mathrm{o_{cl}}$.

Definition 3.5 (Hierarchical Specification Graph) *A* hierarchical specification graph *is a graph $g_{\mathrm{s}}(V_{\mathrm{s}}, E_{\mathrm{s}})$ consisting of a hierarchical problem graph $g_{\mathrm{p}}(V_{\mathrm{p}}, E_{\mathrm{p}})$, an hierarchical architecture graph $g_{\mathrm{a}}(V_{\mathrm{a}}, E_{\mathrm{a}})$, and a set of* mapping edges E_{m}*, where both, the problem as well as the architecture graph are hierarchical graphs according to Definition 3.3.*

The problem and architecture graph are given as hierarchical graphs. The mapping edges $e \in E_{\mathrm{m}}$ map leaf vertices of the problem graph to leaf vertices of the architecture graph. An example of a hierarchical specification graph can be found in Figure 3.8 on page 80.

3. System Synthesis

In this section, we formalize the notion of an *implementation*. An implementation, being the result of a system synthesis, consists of two parts:

1 the *allocation* that indicates which elements of the problem and architecture graph are used in the implementation and

2 the *binding*, i.e., the set of mapping edges which define the binding of vertices in the problem graph to components of the architecture graph.

The term implementation will be used in the same sense as formally defined in [3] but extended here to hierarchical specifications.

3.1 Implementation

An implementation consists of an *allocation* and a *binding*.

Definition 3.6 (Allocation) *An* allocation α *of a given specification graph g_{s} is the subset of all vertices and edges of the problem graph $g_{\mathrm{s}}.g_{\mathrm{p}}$ and the architecture graph $g_{\mathrm{s}}.g_{\mathrm{a}}$ that are used in the implementation, i.e.,*

$$\alpha = \alpha_{\mathrm{v}} \cup \alpha_{\mathrm{e}},$$

where

$$\begin{aligned}\alpha_{\mathrm{v}} &\subseteq \mathrm{V}(g_{\mathrm{s}}.g_{\mathrm{p}}) \cup \mathrm{V}(g_{\mathrm{s}}.g_{\mathrm{a}}) \\ \alpha_{\mathrm{e}} &\subseteq \mathrm{E}(g_{\mathrm{s}}.g_{\mathrm{p}}) \cup \mathrm{E}(g_{\mathrm{s}}.g_{\mathrm{a}})\end{aligned}$$

Here, α_{v} denotes the set of allocated vertices and α_{e} denotes the set of allocated edges.

Definition 3.7 (Binding) *A* binding *β of a given specification graph g_{s} is the subset of mapping edges $g_{\mathrm{s}}.E_{\mathrm{m}}$ used in the implementation, i.e.,*

$$\beta \subseteq g_{\mathrm{s}}.E_{\mathrm{m}}$$

In order to restrict the combinatorial search space, it is useful to determine the set of feasible allocations and feasible bindings.

Definition 3.8 (Feasible Binding) *Given a specification graph g_{s} and an allocation α, a* feasible binding *is a binding β that satisfies the following requirements:*

1 *Each mapping edge $e \in \beta$ starts and ends at an allocated vertex, i.e.,*

$$\forall e = (v_{\mathrm{p}}, v_{\mathrm{a}}) \in \beta : v_{\mathrm{p}}, v_{\mathrm{a}} \in \alpha.$$

2 *For each problem graph vertex $v \in \{\mathrm{V}(g_{\mathrm{p}}) \cap \alpha\}$, exactly one outgoing mapping edge $e \in E_{\mathrm{m}}$ is part of the binding β, i.e.,*

$$|\{e \in \beta \mid e = (v_{\mathrm{p}}, v_{\mathrm{a}}), v_{\mathrm{p}} \in \{\mathrm{V}(g_{\mathrm{p}}) \cap \alpha\} \wedge v_{\mathrm{a}} \in \mathrm{V}(g_{\mathrm{a}})\}| = 1.$$

3 *For each allocated problem graph edge $e \in (v_i, v_j) \in \mathrm{E}(g_{\mathrm{p}}) \cap \alpha$:*

- *either both operations are mapped onto the same vertex, i.e.,*

$$\widetilde{v}_i = \widetilde{v}_j \text{ with } (v_i, \widetilde{v}_i), (v_j, \widetilde{v}_j) \in \beta,$$

- *or there exists an allocated edge $\widetilde{e} = (\widetilde{v}_i, \widetilde{v}_j) \in \{\mathrm{E}(g_{\mathrm{a}}) \cap \alpha\}$ to handle the communication associated with edge e, i.e.,*

$$(\widetilde{v}_i, \widetilde{v}_j) \in \{\mathrm{E}(g_{\mathrm{a}}) \cap \alpha\} \text{ with } (v_i, \widetilde{v}_i), (v_j, \widetilde{v}_j) \in \beta.$$

Definition 3.9 (Feasible Allocation) *A* feasible allocation *is an allocation α that allows at least one feasible binding β.*

Now, we can define an implementation by means of a feasible allocation and feasible binding.

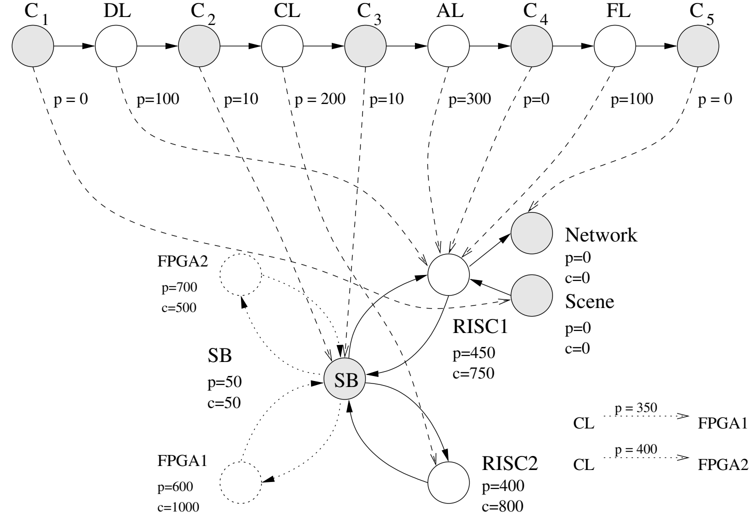

Figure 3.5. Implementation of the MPEG4 coder (see Figure 3.3). All vertices and edges drawn solid describe an allocation. The binding is given by the dashed edges.

Definition 3.10 (Implementation) *Given a hierarchical specification graph g_{s} as defined by Definition 3.5, a* (valid) implementation *is a pair (α, β) where α is a feasible allocation and β is a corresponding feasible binding.*

Example 3.5 Consider the case that the operation CL in Figure 3.3 on page 69 is mapped onto the resource RISC2. Since all other vertices of the problem graph are non-ambiguously bound onto other resources, the dashed mapping edges shown in Figure 3.5 indicate a possible binding. The allocation of vertices is given as:

$$\begin{aligned}\alpha_{\mathrm{v}} \;=\; & \{\mathrm{C}_1, \mathrm{DL}, \mathrm{C}_2, \mathrm{CL}, \mathrm{C}_3, \mathrm{AL}, \mathrm{C}_4, \mathrm{FL}, \mathrm{C}_5, \mathrm{SB}, \\ & \mathrm{RISC1}, \mathrm{RISC2}, \mathrm{Network}, \mathrm{Scene}\}\end{aligned}$$

A feasible binding is given by:

$$\begin{aligned}\beta \;=\; & \{(\mathrm{C}_1, \mathrm{Scene}), (\mathrm{DL}, \mathrm{RISC1}), (\mathrm{C}_2, \mathrm{SB}), (\mathrm{CL}, \mathrm{RISC2}), (\mathrm{C}_3, \mathrm{SB}), \\ & ((\mathrm{AL}, \mathrm{RISC1}), (\mathrm{C}_4, \mathrm{RISC1}), (\mathrm{FL}, \mathrm{RISC1}), (\mathrm{C}_5, \mathrm{Network})\}\end{aligned}$$

Given this allocation and binding, one can see that our implementation is indeed feasible, i.e., α and β are feasible.

3.2 The Task of System Synthesis

With the model introduced previously, the task of system synthesis can be formulated as an optimization problem.

Definition 3.11 (System Synthesis) *The task of* system synthesis *is the following multi-objective optimization problem:*

minimize $o(\alpha, \beta)$,
subject to:
α is a feasible allocation,
β is a feasible binding,
$c_i(\alpha, \beta) \geq 0,\ \forall i \in \{1, \ldots, q\}$.

The constraints on α and β define the set of valid implementations. Additionally, there are functions $c_i, i = 1, \ldots, q$, that determine the set of feasible solutions.

Normally, the *objective function* o is n-dimensional, i.e., we optimize n objectives simultaneously. Furthermore, there are q constraints $c_i, i = 1, \ldots, q$. All possible allocations α and bindings β span the design space X. Only those *design points* $x = (\alpha, \beta) \in X$ that represent a feasible allocation and binding and that satisfy all constraints c_i, are in the set of feasible solutions, or for short in the *feasible set* called $X_{\mathrm{f}} \subseteq X$.

The image of X is defined as $Y = o(X) \subset \mathbb{R}^n$, where the objective function o on the set X is given by $o(X) = \{o(x) \mid x \in X\}$. Analogously, the *objective space* is denoted by $Y_{\mathrm{f}} = o(X_{\mathrm{f}}) = \{o(x) \mid x \in X_{\mathrm{f}}\}$.

Since we are dealing with multi-objective optimization problems, there is generally not only one global optimum, but a set of so-called *Pareto-points* [15]. A Pareto-optimal implementation is a design point which is not worse than any other feasible solution in the design space in all objectives. The *Pareto-set* is the set of all Pareto-optimal solutions.

With this nomenclature, we formally define Pareto-optimality:

Definition 3.12 (Pareto-optimality) *Let α and β be a feasible allocation and a corresponding feasible binding, respectively. A feasible implementation $i = (\alpha, \beta) \in X_{\mathrm{f}}$ is said to be* Pareto-optimal*, if there is no other design point $\widetilde{i} = (\widetilde{\alpha}, \widetilde{\beta}) \in X_{\mathrm{f}}$ which dominates it, i.e.,*

$$\nexists \widetilde{i} \in X_{\mathrm{f}} : \widetilde{i} \succ i,$$

where[1]

$$\begin{array}{rll} i \succ \widetilde{i}\ (i \textit{ dominates } \widetilde{i}) & \textit{iff} & o(i) < o(\widetilde{i}) \\ i \succeq \widetilde{i}\ (i \textit{ weakly dominates } \widetilde{i}) & \textit{iff} & o(i) \leq o(\widetilde{i}) \\ i \sim \widetilde{i}\ (i \textit{ is indifferent to } \widetilde{i}) & \textit{iff} & o(i) \nleq o(\widetilde{i}) \wedge o(i) \ngeq o(\widetilde{i}).^2 \end{array}$$

The set of all Pareto-optimal solutions is called the *Pareto-optimal set*, or the *Pareto-set* X_{p} for short. An approximation of the Pareto-set X_{p} will be termed *quality set* X_{q} in the following.

Example 3.6 An example of a two-dimensional objective space is given in Figure 3.6. Assume that the objectives o_1 and o_2 are both to be minimized.

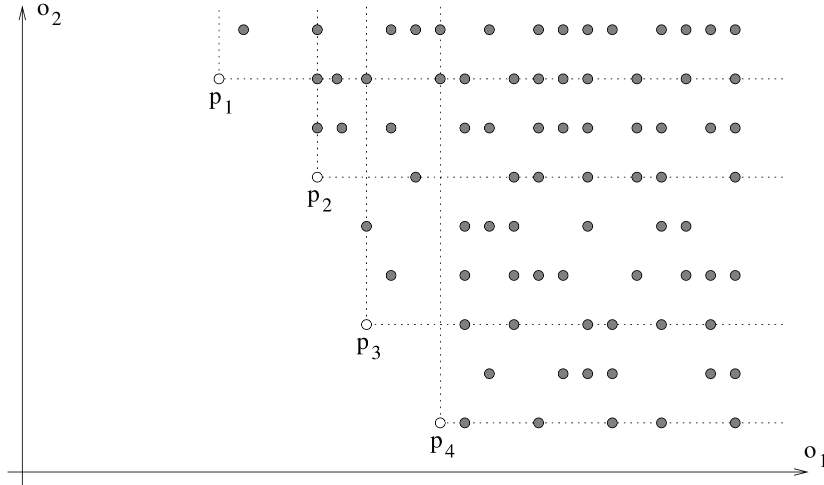

Figure 3.6. Sample objective space of an optimization problem with 2 objectives o_1, o_2 ($n = 2$).

There are four Pareto-optimal points, namely p_1, p_2, p_3, and p_4. All other points are dominated at least by one Pareto-point. The Pareto-set is given by $\{p_1, p_2, p_3, p_4\}$. The remaining points are all dominated by at least one Pareto-optimal solution.

In the following we introduce the notation (g_s, g) to denote a *partial specification* where g is any subgraph in the problem graph $g_s.g_p$ of the given specification g_s.

Definition 3.13 (Partial Specification) *Let $g_s = (V_s, E_s)$ be a specification graph as defined in Definition 3.5 and let g be any subgraph in the corresponding problem graph $g_s.g_p$. A* partial specification (g_s, g) *is obtained by removing all vertices and subgraphs from the problem graph $g_s.g_p$ of the specification graph g_s leaving only g and all its associated subgraphs. To guarantee a meaningful specification, all mapping edges incident to deleted nodes are removed from the specification graph, too. Furthermore, vertices in the architecture graph $g_s.g_a$ of the specification graph g_s which are not incident to any mapping edge, will also be deleted.*

Example 3.7 Given the problem graph shown in Figure 3.4 on page 71. The problem graph of the partial specification $(g_s, \mathrm{Image\&Video})$ is highlighted in Figure 3.7. As described above, all vertices and subgraphs are removed from the problem graph except the subgraph Image&Video and its associated subgraphs (dark shaded subgraphs). In this example, we omitted the architecture graph and mapping edges.

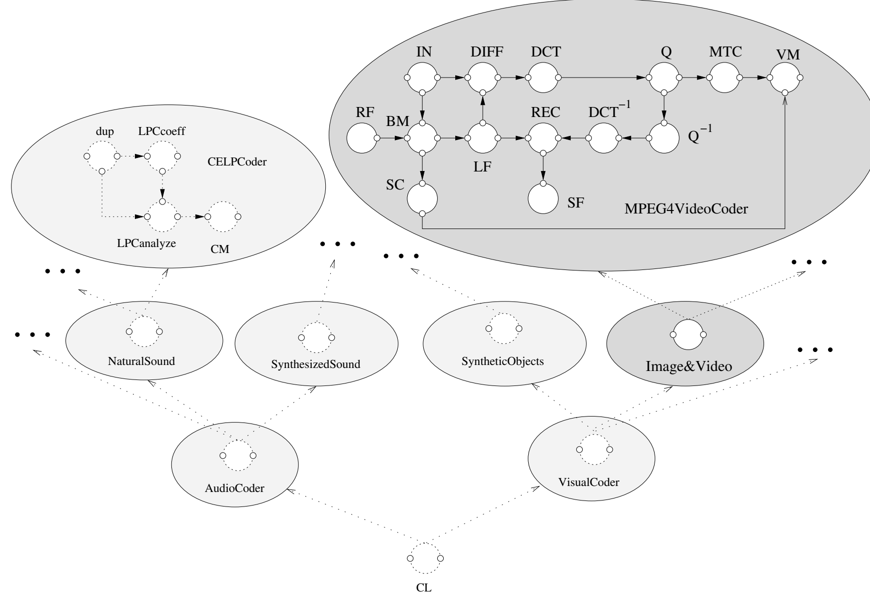

Figure 3.7. Problem graph of a partial specification.

Obviously, the partial specification $(g_{\rm s}, g_{\rm s}.g_{\rm p})$ corresponds to the original specification $g_{\rm s}$. In the following, we denote this particular partial specification as *top-level specification*.

Definition 3.14 (Decomposition) *A* decomposition $\Theta(g_{\rm s}, g_{\rm s}.g_{\rm p})$ *of a top-level specification $g_{\rm s}$ is a partition of $g_{\rm s}$ into partial specifications given by:*

$$\Theta(g_{\rm s}, g_{\rm p}) = \bigotimes_{\forall v \in g_{\rm p}.V : g \in v.G} (g_{\rm s}, g),$$

where $\otimes$ denotes the composition operator which depends on the structure of the problem graph.

Example 3.8 For example, one possible decomposition of the MPEG4 coder shown in Figure 3.7 is given by:

$$\begin{aligned} \Theta(g_{\rm s}, g_{\rm s}.g_{\rm p}) = & \quad (g_{\rm s}, \text{AudioCoder}) \\ & \otimes \ (g_{\rm s}, \text{SyntheticObjects}) \\ & \otimes \ (g_{\rm s}, \text{Image\&Video}) \end{aligned}$$

With the definitions given above, we use the notation $X(g_{\rm s}, g)$ to denote the partial design space regarding the partial specification $(g_{\rm s}, g)$. The respective objective space is given by $Y(g_{\rm s}, g_{\rm p})$. Furthermore, let $X_{\rm p}(g_{\rm s}, g_{\rm p})$ and $X_{\rm f}(g_{\rm s}, g_{\rm p})$

denote the Pareto-set and the feasible set of specification graph g_s, respectively, while meeting the imposed constraints $c = (c_1, c_2, \ldots, c_\mathrm{q})$. $Y_\mathrm{f}(g_\mathrm{s}, g_\mathrm{p})$ is the objective space regarding $X_\mathrm{f}(g_\mathrm{s}, g_\mathrm{p})$.

Hierarchical Design Space Up to now, we only defined optimality and feasibility given the top-level specification. Now, we consider the case of composing a system by solutions of its subsystems preparing the idea of a hierarchical design space exploration. Abraham et al. name three advantages for hierarchical design space exploration techniques [1]:

1 The size of each subsystem's design space is smaller than the top-level design space, i.e.,

$$\forall_{v \in g_\mathrm{p}.V} : \forall_{g \in v.G} : |X(g_\mathrm{s}, g)| \leq |X(g_\mathrm{s}, g_\mathrm{p})| \,.$$

2 The evaluation effort for each subsystem design is low because of the smaller complexity of the subsystem. It is easier to determine $X_\mathrm{f}(g_\mathrm{s}, g_\mathrm{p})$ for small systems, see also [3].

3 The number of top-level design points to be evaluated is a small fraction of the size of the original search space. In our case, this is due to the fact that a valid implementation is only composable of valid solutions of its subsystems (see also item 2), i.e.,

$$X_\mathrm{f}(g_\mathrm{s}, g_\mathrm{p}) \supseteq \bigotimes_{\forall v \in g_\mathrm{p}.V : g \in v.G} X_\mathrm{f}(g_\mathrm{s}, g).$$

Here, $\otimes$ denotes the design space composition operator, which depends on the structure of the problem graph as well as on the chosen objectives. A vertex v whose set of subgraphs is empty ($v.G = \emptyset$), will result in a partial design space composed of all possible bindings β of v and corresponding allocations α.

It seems to be clear that a feasible top-level implementation must be composed of feasible subsystem implementations. Thus, we are able to restrict our search space dramatically. Unfortunately, however, we cannot generally assume that a Pareto-optimal top-level design is composed of Pareto-optimal subsystem implementations. Abraham et al. define necessary and sufficient conditions of the composition function of the objectives which guarantee Pareto-optimality for the top-level system depending on the Pareto-optimality of its subsystems [1]. Applied to our problem of system synthesis, a decomposition $\Theta(g_\mathrm{s}, g_\mathrm{p})$ of a given top-level specification $(g_\mathrm{s}, g_\mathrm{p})$ is called *monotonic* if the top-level Pareto-set $X_\mathrm{p}(g_\mathrm{s}, g_\mathrm{p})$ is given by the composition $\otimes$ of the Pareto-sets $X_\mathrm{p}(g_\mathrm{s}, g_i)$ for all subsystems $(g_\mathrm{s}, g_i) \in \Theta(g_\mathrm{s}, g_\mathrm{p})$. This is true if the composition function of each

objective is a monotonic function. In that case, we would be able to construct the true Pareto-front from the Pareto-optimal solutions of its subsystems.

Although this observation is important and interesting, the optimization goals in embedded system design unfortunately do not possess these monotonicity properties. In fact, they depend on the system composition operator $\otimes$ as will be shown in the next section. The major question therefore is whether we have to combine all feasible implementations of all subsystems or whether it is possible to find the Pareto-optimal solutions (or a good approximation thereof) by exploring a reduced design space. This directly leads to two different optimization approaches proposed in this chapter (Section 5). Before we present these ideas, we formally define the most important objectives for embedded system design and used throughout this chapter.

3.3 Objective Space

In this section, we consider a three-dimensional objective space which is defined by the *cost*, the overall *power consumption* and the *flexibility* of an implementation.

Definition 3.15 (Implementation Cost (Objective 1)) *The* implementation cost $\mathrm{cost}(i)$ *for a given implementation* $i = (\alpha, \beta)$ *is given by the sum of cost of all allocated resources* $v \in \alpha \cap V(g_\mathrm{a})$*, i.e.,*

$$\mathrm{cost}(i) = \sum_{v \in \alpha \cap V(g_\mathrm{a})} \mathrm{cost}(v)$$

Example 3.9 Again, we look at Example 3.5 on page 74. In order to calculate the implementation cost $\mathrm{cost}(i)$, we have to compute the set of allocated resources, i.e.,

$$\alpha_\mathrm{v} \cap V(g_\mathrm{a}) = \{\mathrm{RISC1}, \mathrm{SB}, \mathrm{RISC2}, \mathrm{Network}, \mathrm{Scene}\}$$

Now, we calculate the implementation cost of this implementation i:

$$\begin{aligned}\mathrm{cost}(i) &= \mathrm{cost}(\mathrm{RISC1}) + \mathrm{cost}(\mathrm{RISC2}) + \mathrm{cost}(\mathrm{SB}) + \\ &\quad\ \mathrm{cost}(\mathrm{Network}) + \mathrm{cost}(\mathrm{Scene}) \\ &= 750 + 800 + 50 + 0 + 0 = 1600\end{aligned}$$

With this definition, we are able to show that unfortunately, the composition function of the implementation cost is a non-monotonic function.

Theorem 3.1 *The composition function for the cost objective of an embedded system is non-monotonic.*

Proof 3.1 *We prove this theorem by contradiction using an example.*

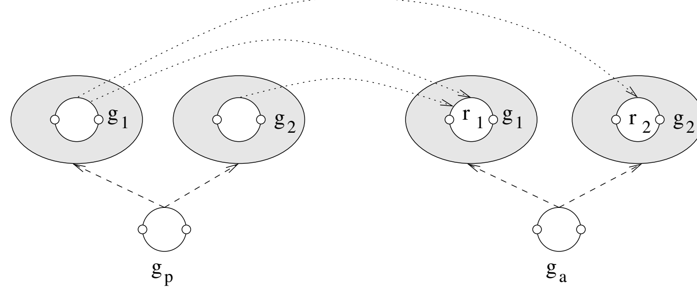

Figure 3.8. Sample specification.

Given the specification graph g_s *depicted in Figure 3.8. Let the allocation cost for* r_1 *and* r_2 *be 200 and 100, respectively. The cost objective for the Pareto-optimal design regarding the cost of allocated resources of the subsystems are given by* $\mathrm{cost}(X_\mathrm{p}(g_\mathrm{s}, g_1)) = \{(100)\}$ *and* $\mathrm{cost}(X_\mathrm{p}(g_\mathrm{s}, g_2)) = \{(200)\}$. *When considering subsystem* (g_s, g_1) *alone, we would allocate resource* r_1 *as a cost-minimal implementation. When considering subsystem* (g_s, g_2) *alone, the resource* r_2 *would determine its cost-optimal implementation. By combining both implementations, we obtain an implementation of the top-level design* $(g_\mathrm{s}, g_\mathrm{p})$*. Due to the allocation of both resources, we get implementation cost of* $\mathrm{cost}(X_\mathrm{p}(g_\mathrm{s}, g_1)) \otimes \mathrm{cost}(X_\mathrm{p}(g_\mathrm{s}, g_2)) = 300$ *for the combined implementation which is obviously suboptimal as* $\mathrm{cost}(X_\mathrm{p}(g_\mathrm{s}, g_\mathrm{p})) = 200$.

■

If each subsystem is bound onto a dedicated resource, we could sum up the cost of each subsystem in order to calculate the overall cost. Then, if we could find a single resource with smaller cost, we could also optimize our top-level design. The most meaningful computation of the overall implementation cost of a top-level design while sharing resources among its subsystems is given by the sum of all *used* resources. Hence, the definition of cost optimal implementations of subsystems will not provide cost optimal solutions at higher hierarchical levels.

Our second objective, the overall power consumption is the sum of the power consumptions of all allocated resources and the additional power consumptions originating by binding processes to resources.

Definition 3.16 (Power Consumption (Objective 2)) *The overall* power consumption $\mathrm{power}(i)$ *of a given implementation* $i = (\alpha, \beta)$ *is given by the sum of power consumption of all allocated resources* $v \in \alpha \cap V(g_\mathrm{a})$ *plus the additional power consumption annotated at the mapping edges* $e \in \beta$*, i.e.,*

$$\mathrm{power}(i) = \sum_{v \in \alpha \cap V(g_\mathrm{a})} \mathrm{power}(v) + \sum_{e \in \beta} \mathrm{power}(e)$$

Example 3.10 The implementation described in Example 3.5 possesses the following overall power consumption:

$$\begin{aligned}\text{power}(i) &= \text{power}(\text{RISC1}) + \text{power}(\text{RISC2}) + \text{power}(\text{SB}) + \\ &\quad \text{power}(\text{Network}) + \text{power}(\text{Scene}) + \\ &\quad \text{power}((\text{C}_1, \text{Scene})) + \text{power}((\text{DL}, \text{RISC1})) + \\ &\quad \text{power}((\text{C}_2, \text{SB})) + \text{power}((\text{CL}, \text{RISC2})) + \\ &\quad \text{power}((\text{C}_3, \text{SB})) + \text{power}((\text{AL}, \text{RISC1})) + \\ &\quad \text{power}((\text{C}_4, \text{RISC1})) + \text{power}((\text{FL}, \text{RISC1})) + \\ &\quad \text{power}((\text{C}_5, \text{Network})) \\ &= 450 + 400 + 50 + 0 + 0 + 0 + 100 + 10 + 200 + \\ &\quad 10 + 300 + 0 + 100 + 0 = 1620\end{aligned}$$

The third objective, the reciprocal of a system's flexibility, is defined as follows:

Definition 3.17 (Flexibility (Objective 3)) *The* flexibility $f(g)$ *of a given problem (sub)graph* g *is expressed as:*

$$f(g) = \alpha(g) \cdot \begin{cases} \left[\sum_{v \in g.V} \sum_{\widehat{g} \in v.G} f(\widehat{g})\right] - |\{v \mid v \in g.V \wedge |v.G| \geq 1\}| + 1 \\ \qquad \textit{for } \{v \mid v \in g.V \wedge |v.G| \geq 1\} \neq \emptyset \\ 1 \qquad \textit{otherwise} \end{cases}$$

where the term $\alpha(g)$ *describes the utilization* ($\alpha(g) = 1$) *of the subgraph* g *in the implementation, otherwise* $\alpha(g) = 0$*.*

In other words: The flexibility of a subgraph g, if element of the allocation, is calculated by the sum of the flexibilities of all its vertices minus the number of hierarchical vertices less 1, and 1 if there is no hierarchical vertex in the given subgraph. The flexibility of a hierarchical vertex is the sum of flexibilities of all its associated subgraphs. If a subgraph is not in the implementation, its flexibility is 0.

For a comprehensive illustration of a system's flexibility, see [12]. Without loss of generality, we only treat system specifications throughout this chapter where the maximal flexibility equals the number of leaf graphs.

4. System Synthesis Using Evolutionary Algorithms

An Evolutionary Algorithm (EA) is characterized by the fact that a number N of potential solutions (called *individuals* $j \in X$, where X represents the space of all possible individuals) of the optimization problem simultaneously sample the search space. This *population* $P = \{j_1, j_2, \cdots, j_N\}$ is modified according to the natural evolutionary process: after initialization, selection and recombination are executed in a loop for a fixed number of iterations. Each run of the loop is called a *generation* and P_t denotes the population at generation t.

The selection operator is intended to improve the average quality of the population by giving individuals of higher quality a higher probability of survival. Selection thereby focuses the search on promising regions in the search space. The quality of an individual is measured by a fitness function. Recombination changes the genetic material in the population either by crossover or by mutation in order to explore new points in the search space. Depending on the problem to be solved, various codings for EAs exist, e.g., the individuals are represented by bit strings, vectors of integers or reals, trees, graphs. The choice of coding determines also the recombination operator.

4.1 Multi-Objective Evolutionary Algorithms

Evolutionary Algorithms in multi-objective optimization (MOEAs) are investigated by many research groups. We can divide these methods in two groups, Elitist MOEA and Non-elitist MOEA. In Elitist MOEA, the best solutions of each population are kept in an archive. In fact, the presence of elites enhances the probability of creating better offsprings and it has been proven [16] that a class of Evolutionary Algorithms converge to the global optimal solution of some functions in the presence of elitism. Elitist MOEAs are investigated in algorithms like NSGA2, PAES, Rudolph and Agapie's Elitist, SPEA2, etc. [5]. In the following we shortly review some of these algorithms, then we will introduce SPEA2 [19] that serves as the basis of both of our optimization approaches to system synthesis.

Rudolph and Agapie's Elitist Genetic Algorithm (AR1). Rudolph and Agapie suggested a Multi-Objective Evolutionary Algorithm (MOEA) which uses elitism [17]. In its general format, μ parents are used to create λ ($\lambda \geq \mu$) offsprings using genetic operators. So in each generation, there are two populations, the parent population P_t and the offspring population Q_t. The algorithm works in three phases. In the first phase, non-dominated solutions of Q_t are identified, deleted from Q_t, and placed into an elite population $\overline{P}_t$. In the second phase, each solution q of $\overline{P}_t$ is compared with each solution of the parent population P_t. If q dominates a solution in P_t, that solution is deleted from P_t and q is moved into a set P'_t. On the other hand, if q is indifferent to the solutions in P_t, we remove it from $\overline{P}_t$ and put it into another set Q'_t. In the third phase, all the above sets are arranged in a special order of performance. First, P_t and P'_t are combined. If they together do not fill up the whole population, we take solutions from the the sets in following order: Q'_t, $\overline{P}_t$ and Q_t. This algorithm guarantees convergence to the true Pareto-optimal front. A disadvantage of this algorithm is that it doesn't guarantee a good distribution of diverse solutions on the Pareto-optimal front.

Elitist Non-dominated Sorting Genetic Algorithm (NSGA2). In this algorithm [4], the offspring population Q_t of size N is created from the parent population P_t of size N. First, these two populations are combined to form a population R_t of size $2N$. Then, a non-dominated sorting procedure is used to classify the entire population R_t as follows: Non-dominated individuals are calculated from R_t and are called non-dominated solutions of front 1. Then, these are temporarily disregarded from R_t and the non-dominated solutions of the remaining elements of R_t are then determined and called non-dominated solutions of front 2. This procedure is continued until all the members of R_t are classified into a non-dominated front. After this sorting process, the new population is filled by solutions of different non-dominated fronts, one at a time. The filling starts with the best non-dominated front and so on. Since R_t has $2N$ solutions and the new population must have N solutions, just N best solutions appear in the new population (N solutions from the best fronts). In the case of inadequate available space in the new population to accommodate all solutions of a non-dominated set, a crowding strategy is used to identify solutions which reside in less crowded areas.

Pareto-Archived Evolution Strategy (PAES). In this algorithm, each child is compared with its parents. If the child dominates a parent, the child is accepted as a parent in the next generation [14]. If a parent dominates the child, the child is discarded and a new child must be created. In the case that both are indifferent, the child will be compared with an archive of so far best solutions. If it dominates any member in the archive, it will be a parent in the next generation. But if the child does not dominate any member in the archive, both parent and child are compared for their nearness (distance) to members of the archive. If the child resides in a least crowded region, it is accepted as a parent and will be added to the archive. In the case that child and parent have the same nearness (distance) to the archive, one of them is selected at random.

4.2 Strength Pareto Evolutionary Algorithm

SPEA2 in contrast to its predecessor SPEA (Strength Pareto Evolutionary Algorithm, [21] and [19]) incorporates a fined-grained fitness assignment strategy, a density estimation technique, and an enhanced truncation method that are explained in the following. This method uses Pareto-dominance based selection [10] and elitism where the policy is to always include the $\overline{N}$ best individuals of the current generation into the next generation in order not to loose them during exploration. These $\overline{N}$ best individuals are stored in an archive $\overline{P}_t$. Finally, in order to maintain a high diversity within the population, it uses an enhanced truncation method, which guarantees a high diversity of individuals in the archive. Altogether, SPEA2 has shown to provide superior results compared to existing approaches, see, e.g., [19] for many problems of interest. The

algorithm of SPEA2 as it will be used here for multi-objective optimization is listed below:

SPEA2 ALGORITHM

IN: N: population size, $\overline{N}$: Archive size
T: Maximum number of generations

OUT: $\overline{P}$: Non-dominated set

Step1: *Initialization:* Initial population P_t and an empty archive $\overline{P}_t$.
Set t as the current generation and $t = 0$.

Step2: *Fitness Assignment:* Calculate fitness values of individuals in P_t and $\overline{P}_t$.

Step3: *Environmental Selection:* Copy all non-dominated individuals in P_t and $\overline{P}_t$ to $\overline{P}_{t+1}$. If size of $\overline{P}_{t+1}$ exceeds $\overline{N}$, then reduce $\overline{P}_{t+1}$ by means of the truncation operator, otherwise if size of $\overline{P}_{t+1}$ is less than $\overline{N}$, then fill $\overline{P}_{t+1}$ with dominated individuals in $\overline{P}_t$ and P_t.

Step4: *Termination:* If $t > T$ or another stopping criterion is satisfied, then set $\overline{P}$ to the set of decision vectors represented by $\overline{P}_{t+1}$.

Step5: *Mating Selection:* Perform binary tournament selection with replacement on $\overline{P}_{t+1}$ in order to fill the mating pool.

Step6: *Variation:* Apply recombination and mutation operators to the mating pool and set P_{t+1} to the resulting population.
Increment generation counter ($t = t + 1$) and go to step 2.

Fitness Assignment. In SPEA2 [19], both dominating and dominated solutions are taken into account for calculating the fitness value of each individual (to avoid the situation that individuals dominated by the same archive members get identical fitness values). In detail, to each individual j in the archive $\overline{P}_t$ and the population P_t is assigned a strength value $S(j)$, representing the number of solutions it dominates:

$$S(j) = \left| \left\{ \widetilde{j} \mid \widetilde{j} \in P_t + \overline{P}_t \wedge j \succ \widetilde{j} \right\} \right| \tag{3.1}$$

Here, $+$ stands for multi set union. On the basis of the strength S, the raw fitness $R(j)$ of an individual j is calculated:

$$R(j) = \sum_{\widetilde{j} \in P_t + \overline{P}_t, \widetilde{j} \succ j} S(\widetilde{j}) \tag{3.2}$$

For Example, Figure 3.9 shows the raw fitness values for a given population and a minimization problem with two objectives o_1 and o_2.

Additional density information is incorporated to discriminate between individuals having identical raw fitness values. The density estimation technique

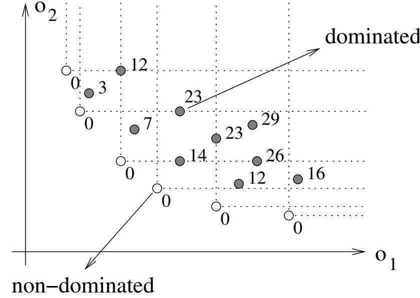

Figure 3.9. Raw fitness values for a minimization problem with two objectives o_1 and o_2 using SPEA2.

used in SPEA2 is an adaptation of a kth nearest neighbor method, where the density at any objective vector j is a function $D(j)$ of the distance to the kth nearest objective vector. The final fitness value will be $F(j) = R(j) + D(j)$. In this case, the final fitness values of the non-dominated points are less than 1 and not zero anymore.

Truncation Method. In this step which is also called environmental selection, the first step is to copy all non-dominated individuals, i.e., those which have a fitness lower than one, from archive and population to the archive of the next generation:

$$\overline{P}_{t+1} = \left|\{j \mid j \in P_t + \overline{P}_t \wedge F(j) < 1\}\right| \tag{3.3}$$

where $F(j)$ is the fitness value of j. If the non-dominated front fits exactly into the archive, the environmental selection step is completed. Otherwise, there can be two situations: Either the archive is too small or too large. In the first case, the best dominated individuals in the previous archive and population are copied to the new archive. In the second case, the archive must be truncated and some of the individuals in the archive must be deleted. In SPEA2, the individual which has the minimum objective distance to another individual is chosen at each stage; if there are several individuals with minimum distance, the tie is broken by considering the second smallest distance, and so forth.

4.3 Chromosome Structure for System Synthesis

First, we explain a suitable chromosome structure of an EA for non-hierarchical specification graphs. Later, we propose a hierarchical extension of this structure. The explanation of the non-hierarchical chromosome structure, however, is needed first. Each individual consists of two components, an *allocation* and a *binding*, as defined in Definition 3.6 and 3.7 (see Figure 3.10).

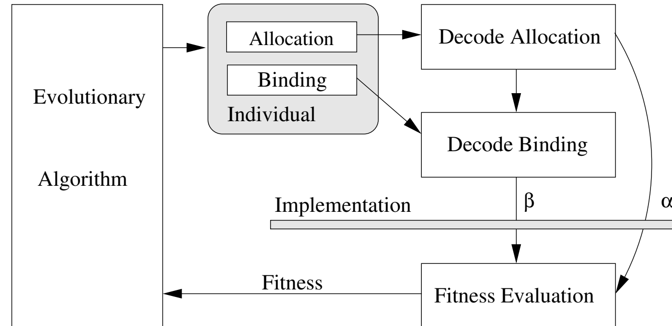

Figure 3.10. The decoding of an individual to an implementation.

To obtain a meaningful coding for the task of system synthesis, one has to address the question of how to handle infeasible allocations and infeasible bindings suggested by the EA. Obviously, if allocations and bindings may be randomly chosen, a lot of them can be infeasible. In general, there are two different methods to handle these infeasible implementations: Punishing and Repairing. Here, repairing is used. Because of the well known properties of feasible allocations and bindings, one can "repair" infeasible individuals by incorporating domain knowledge in these repair mechanisms easily. But as the determination of a feasible allocation or binding is NP-complete [3], this would result in solving an NP-complete task for each individual to be repaired.

These considerations have led to the following compromise: The randomly generated allocations of the EA are partially repaired using a heuristic. Possible complications detected later on during the calculation of the binding will be considered by a penalty. Hence, the mapping task can be divided in three steps:

1 First, the allocation of resources $v \in V_{\mathrm{a}}$ is decoded from the individual and repaired with a simple heuristic (the function *allocation()*),

2 Next the binding of the edges $e \in E_{\mathrm{m}}$ is performed (the function *binding()*), and

3 Finally, the allocation is updated in order to eliminate unnecessary vertices $v \in V_{\mathrm{a}}$ from the allocation and all necessary edges $e \in E_{\mathrm{a}}$ are added to the allocation (the function *update_allocation()*).

DECODING
IN: The individual j consisting of allocation $alloc$,
repair allocation priority list L_{R}, binding order list L_{O},
and binding priority list $L_{\mathrm{B}}(v)$.
OUT: The allocation α and the binding β if both are feasible,
$(\{\},\{\})$ if no feasible binding is represented by the individual j

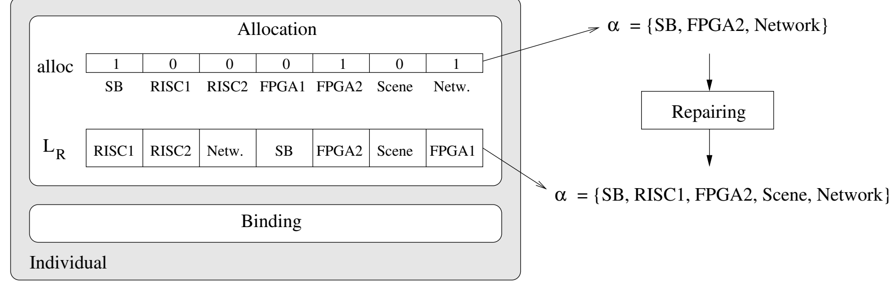

Figure 3.11. An example of the coding of an allocation.

```
BEGIN
  ᾱ ← allocation(alloc(j), L_R(j))
  β̄ ← binding(L_B(j), L_O(j), ᾱ)
  IF β̄ = {}
    RETURN ({}, {})
  ENDIF
  β ← β̄
  α ← update_allocation(ᾱ, β̄)
  RETURN (α, β)
END
```

One iteration of the loop results in a feasible allocation and binding of the vertices and edges of the problem graph $g_{\rm p}$ to the vertices and edges of the architecture graph $g_{\rm a}$. If no feasible binding could be found, the whole decoding of the individual is aborted.

In the following the functions *allocation()*, *binding()*, and *update-allocation()* are explained in detail.

4.3.1 The Function allocation(). The allocation of vertices is directly encoded in the chromosome, i.e., the elements in a vector $alloc$ encode for each vertex $v \in V_{\rm a}$ if it is activated or not, i.e., $a(v) = alloc[v]$. This simple coding might result in many infeasible allocations. Due to this fact, a simple repair heuristic is applied. This heuristic only adds new vertices $v \in V_{\rm a}$ to the allocation and reflects the simplest case of infeasibility that may arise from non-executable functional vertices: Consider the set $V_{\rm B} \subseteq V_{\rm p}$ that contains all vertices that can not be executed, because not a single corresponding resource vertex is allocated, i.e., $V_{\rm B} = \{v \in V_{\rm p} \mid \forall \widetilde{v} \in V_{\rm a} : (v, \widetilde{v}) \in E_{\rm m} \wedge a(\widetilde{v}) = 0\}$. To make the allocation feasible (in this sense) for each $v \in V_{\rm B}$, at most one $\widetilde{v} \in V_{\rm a}$ is added, until feasibility in the sense above is achieved.

```
ALLOCATION
  IN:   The allocation alloc and repair allocation
        priority list L_R of individual j
  OUT:  The allocation α
  BEGIN
    α ← {}
    FORALL ṽ ∈ V_a DO
      IF (alloc[ṽ] = 1)
        α ← α ∪ {ṽ}
      ENDIF
    ENDFOR
    V_B ← not_bindable_nodes(α)
    ṽ_r ← first(L_R)
    WHILE (V_B ≠ {}) DO
      IF (V_B ≠ not_bindable_nodes(α ∪ {ṽ_r})
        α ← α ∪ {ṽ_r}
        V_B ← not_bindable_nodes(α)
      ENDIF
      ṽ_r ← next(L_R)
    ENDWHILE
    RETURN α
  END
```

The order in which additional resources are added has a large influence on the resulting allocation. For example, one could be interested in an additional allocation with minimal cost. As this depends on the optimization goal expressed in the objective function o, the order should automatically be adapted (see also Figure 3.11). This will be achieved by introduction of a *repair allocation priority list* L_{R} coded in the individual. In this list, all resources $v \in V_{\mathrm{a}}$ are contained and the order in the list determines the order the vertices will be added to the allocation. This list also undergoes genetic operators like crossover and mutation and can therefore be optimized by the Evolutionary Algorithm.

Example 3.11 Consider the specification of Example 3.3 in Figure 3.3. In Figure 3.11, the allocation information as stored in the individual is shown on the left. The direct decoding of the allocation string yields the allocation $\alpha = \{\mathrm{SB}, \mathrm{FPGA2}, \mathrm{Network}\}$. This allocation is not valid as there exists no allocated resource for executing $\mathrm{C}_1, \mathrm{DL}, \mathrm{AL}, \mathrm{C}_4, \mathrm{FL} \in g_p$. This allocation is then repaired using the repair allocation priority list. $\mathrm{SB}, \mathrm{FPGA2}$ and $\mathrm{Network}$ are already in the allocation. So $\mathrm{RISC1}$ and Scene are allocated. The rest of the list is then ignored, as no node remains unmappable.

4.3.2 The Function binding().

A binding is obtained by activating exactly one edge $e \in E_{\mathrm{m}}$ adjacent to an allocated vertex v for each $v \in V_{\mathrm{p}}$. The problem of coding the binding lies in the strong inter-dependence of the binding and the current allocation. As crossover or mutation might change the allocation, a directly encoded binding could be meaningless for a different allocation. Hence, a coding of the binding is of interest that can be interpreted *independently* of the allocation. This is achieved in the following way:

For each problem graph vertex $v \in V_{\mathrm{p}}$, a list is coded as allele that contains all adjacent edges $e \in E_{\mathrm{m}}$ of v. This list is seen as a priority list and the first edge e_k with $e_k = (v, \widetilde{v})$ that gives a feasible binding is included in the binding, i.e., $\alpha(e_k) := 1$. The test of feasibility is directly related to the definition of a feasible binding (see Definition 3.8). Details are given in the following algorithms. Note that it is possible that no feasible binding is specified by the individual. In this case, β is the empty set, and the individual will be given a penalty value as its fitness value.

```
BINDING
  IN:   The binding priority lists L_B(v) ∀v ∈ V_p, the
        binding order list L_O, and the allocation α of an individual j.
  OUT:  The binding β, or {} if no feasible binding was decoded.
  BEGIN
    β ← {}
    FORALL u ∈ L_O ∩ α DO
      e' ←null
      FORALL e ∈ L_B(u) ∩ α do
        IF (is_feasible_binding(e, β, α))
          e' ← e
          BREAK
        ENDIF
      ENDFOR
      IF (e' = null)
        RETURN {}
      ELSE
        β ← β ∪ {e'}
      ENDIF
    ENDFOR
    RETURN β
  END
```

Example 3.12 Figure 3.12 shows an example of a binding as it is coded in an individual allocated in Example 3.3 (Figure 3.3). The binding order specified

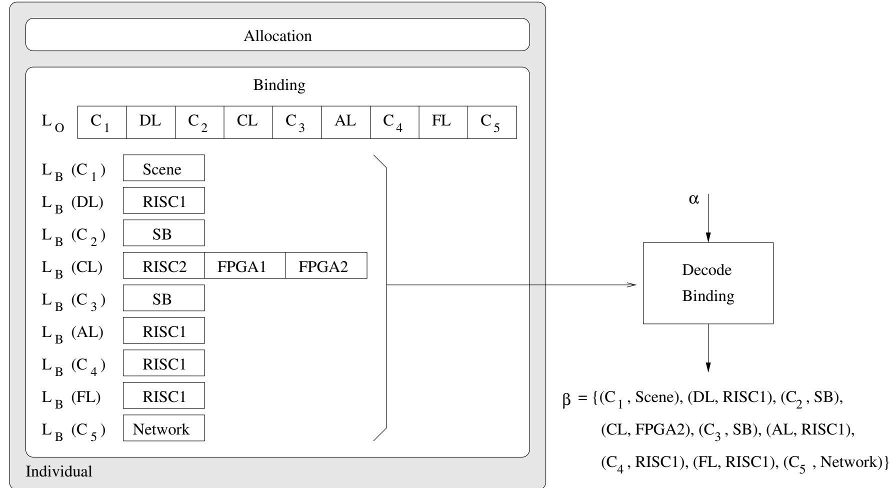

Figure 3.12. An example of the coding of a binding.

by the list L_{O} is $\mathrm{C}_1, \mathrm{DL}, \mathrm{C}_2, \mathrm{CL}, \mathrm{C}_3, \mathrm{AL}, \mathrm{C}_4, \mathrm{FL}, \mathrm{C}_5$. The binding priority lists for all nodes are also shown. For example, the priority list for node CL implies to bind CL to the resource RISC2. If this is not possible, it should be bound to FPGA1, and if this is also not possible, it should be bound to FPGA2. As the allocation from Example 3.11 does not contain RISC2 and FPGA1, this node is finally bound to FPGA2.

Finally, in the function update-allocation, vertices of the allocation that are not used will be removed from the allocation. Furthermore, all edges $e \in E_{\mathrm{a}}$ in the architecture graph g_{a} are added to the allocation that are necessary to obtain a feasible binding.

5. Hierarchical Design Space Exploration

In the previous section, an EA coding for allocations and bindings of non-hierarchical specification graphs was revised. Here, we want to present EA-based techniques for hierarchical graphs. For this purpose, two novel approaches for design space exploration of hierarchically structured specifications are proposed.

1 Pareto-Front Arithmetics

2 Hierarchical Chromosomes

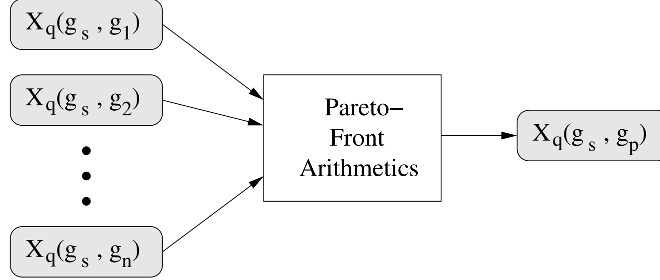

Figure 3.13. Concept of Pareto-front arithmetics.

5.1 Pareto-Front Arithmetics

In this section, we introduce a hierarchical design space exploration scheme called *Pareto-front arithmetics* in the context of hierarchical system synthesis. The main idea is to compute a quality set of a top-level design from the Pareto-sets (or a quality set) of the subsystems. This hierarchical construction of the quality set will be termed *Pareto-front arithmetics*.

Figure 3.13 explains this concept of Pareto-front arithmetics. The inputs are the quality sets of the mutual exclusive partial specifications (g_{s}, g_1), (g_{s}, g_2), $\ldots, (g_{\mathrm{s}}, g_n)$, where (g_{s}, g_1), (g_{s}, g_2) are mutual exclusive iff $g_1 \cap g_2 = \emptyset$. Here, we can consider each partial specification as a non-hierarchical specification associated with a leaf graph of the problem graph. In order to optimize such a partial specification, we use the MOEA SPEA2 as described in the previous section. The resulting quality sets are then used as inputs for Pareto-front arithmetics (see Figure 3.13). The result is a quality set of the top-level specification $(g_{\mathrm{s}}, g_{\mathrm{s}}.g_{\mathrm{p}})$.

Figure 3.14 shows three operations of how to possibly combine Pareto-fronts of subsystems that may be used by Pareto-front arithmetics: The first operation (Figure 3.14(a)) is the union of two or more Pareto-fronts, i.e., each Pareto-optimal solution is added to the resulting set. All points not dominated in the resulting set determine the Pareto-set of the next higher hierarchical level.

The second operation is to take the maximum of each objective in order to combine two (or more) points (Figure 3.14(b)). Here, each Pareto-optimal point $p_1 \in \{\mathrm{p}_{11}, \mathrm{p}_{12}, \mathrm{p}_{13}, \mathrm{p}_{14}\}$ is combined with each Pareto-optimal solution $p_2 \in \{\mathrm{p}_{21}, \ldots, \mathrm{p}_{25}\}$. The resulting objectives $(\mathrm{o}_1, \mathrm{o}_2)$ of the composed solutions are the maxima of the respective subsystems' objectives, i.e., $\mathrm{o}_1(p_{3x}) = max(\mathrm{o}_1(p_{1i}), \mathrm{o}_2(p_{2j}))$, and $\mathrm{o}_2(p_{3x}) = max(\mathrm{o}_2(p_{1i}), \mathrm{o}_2(p_{2j}))$. The composed solutions are filtered regarding Pareto-optimality. Figure 3.14(b) shows six resulting Pareto-points.

Figure 3.14(c) outlines the addition of the objective of two or more Pareto-points. Again, each Pareto-optimal solution $p_1 \in \{\mathrm{p}_{11}, \ldots, \mathrm{p}_{14}\}$ is com-

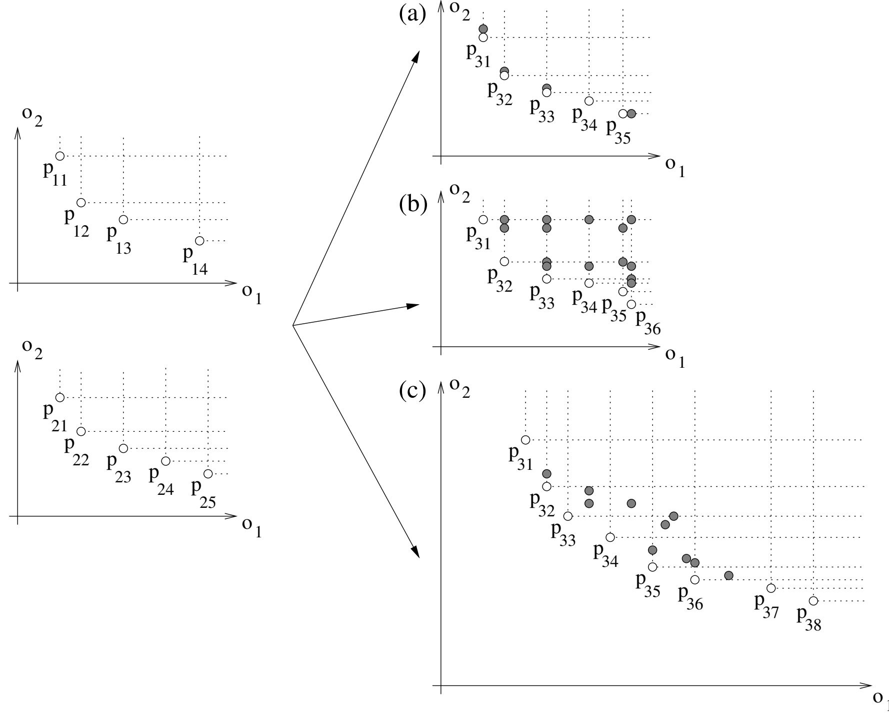

Figure 3.14. Example Pareto-front arithmetics operations (a) union, (b) maximum, and (c) addition of objectives of Pareto-points.

bined with each point $p_2 \in \{\mathrm{p}_{21}, \ldots, \mathrm{p}_{25}\}$. Here, the resulting objectives are calculated by the sum of the objectives of the subsystems, i.e., $\mathrm{o}_k(p_{3x}) = \mathrm{o}_k(p_{1i}) + \mathrm{o}_k(p_{2j})$ for $k = 1, 2$. Since the presented operations are all monotonic, the resulting optimality sets are indeed Pareto-sets, see [1].

The objectives used here for optimizing an embedded system, namely cost, power, and flexibility, are more complex due to resource sharing, power consumption being dependent on the binding, etc. Consequently, the operations for Pareto-front arithmetics are more sophisticated as the operations shown in Figure 3.14. Furthermore, these operations are non-monotonic as has been shown for the objective implementation cost in Section 3. Hence, we cannot claim Pareto-optimality for the implementations in the resulting optimality set when using Pareto-front arithmetics in general.

More formally, Pareto-front arithmetics operates in the objective space, i.e.,

$$\mathrm{o} = \mathrm{h}(y_1, y_2, \ldots, y_n), \text{ where } y_j = \mathrm{o}(x_j) \; \forall 1 \leq j \leq n.$$

In Section 6.3 we will present a case-study where Pareto-front arithmetics is applied to calculate an approximation of the Pareto-set. In order to compute the

objectives for the next hierarchical level, we use the following approximation (addition of objectives). In particular, the performed operations are:

$$\begin{aligned}
\mathrm{cost}((g_\mathrm{s}, g_i)), (g_\mathrm{s}, g_j)) &\approx \mathrm{cost}((g_\mathrm{s}, g_i)) + \mathrm{cost}((g_\mathrm{s}, g_j)) \\
\mathrm{power}((g_\mathrm{s}, g_i)), (g_\mathrm{s}, g_j)) &\approx \mathrm{power}((g_\mathrm{s}, g_i)) + \mathrm{power}((g_\mathrm{s}, g_j)) \\
\mathrm{f}((g_\mathrm{s}, g_i)), (g_\mathrm{s}, g_j)) &\approx \mathrm{f}((g_\mathrm{s}, g_i)) + \mathrm{f}((g_\mathrm{s}, g_j))
\end{aligned}$$

5.2 Hierarchical Chromosomes

An alternative approach is to consider an EA that exploits the hierarchical structure of the specification by encoding the structure in the chromosome itself. In order to extend the presented approaches, we have to capture the hierarchical structure of the specification in our chromosomes first. Figure 3.15 gives an example for such a chromosome.

The depicted chromosome encodes an implementation of the problem graph first introduced in Figure 3.4. The underlying architecture graph is the one shown in Figure 3.2. The structure of the chromosome in Figure 3.15 resembles the structure of the problem graph given in Figure 3.4. The leaves of the chromosome are nearly identical to the non-hierarchical chromosome structure described in Section 4 except for the lack of the allocation and allocation repairing list. These have moved to the top-level node of the chromosome (see Figure 3.15).

Furthermore, the chromosome is constructed by using hierarchical nodes. Each hierarchical node resembles a subgraph of the underlying problem graph. For each hierarchical vertex v in the corresponding subgraph g, the hierarchical node contains a selection list L_S. Each entry in this list describes the use of subgraph $\widetilde{g} \in v.G$ associated with v in the implementation. Only leaves that are selected all the way down from the top-level node are termed *allocated leaves*. A leaf which is not selected is named a *non-allocated leaf.* If we encounter a subgraph that does not include hierarchical vertices, we encode it by using a non-hierarchical chromosome as described above.

Since each subgraph, if used in the implementation, has to be executed on the same architecture, we store the global allocation string and a global allocation repairing list in the top-level node. Despite the modified internal structure, our hierarchical chromosome resembles exactly the non-hierarchical chromosome by still encoding an allocation and a binding. The main difference lies in the variable number of problem graph vertices allocated and bound in any given individual.

We therefore still can use the same evolutionary operations, namely mutation and crossover, as well as the function *allocation()* that have been introduced for the non-hierarchical structure (Section 4.3). However, we propose two additional generic operators making use of the hierarchical structure of the chromosome:

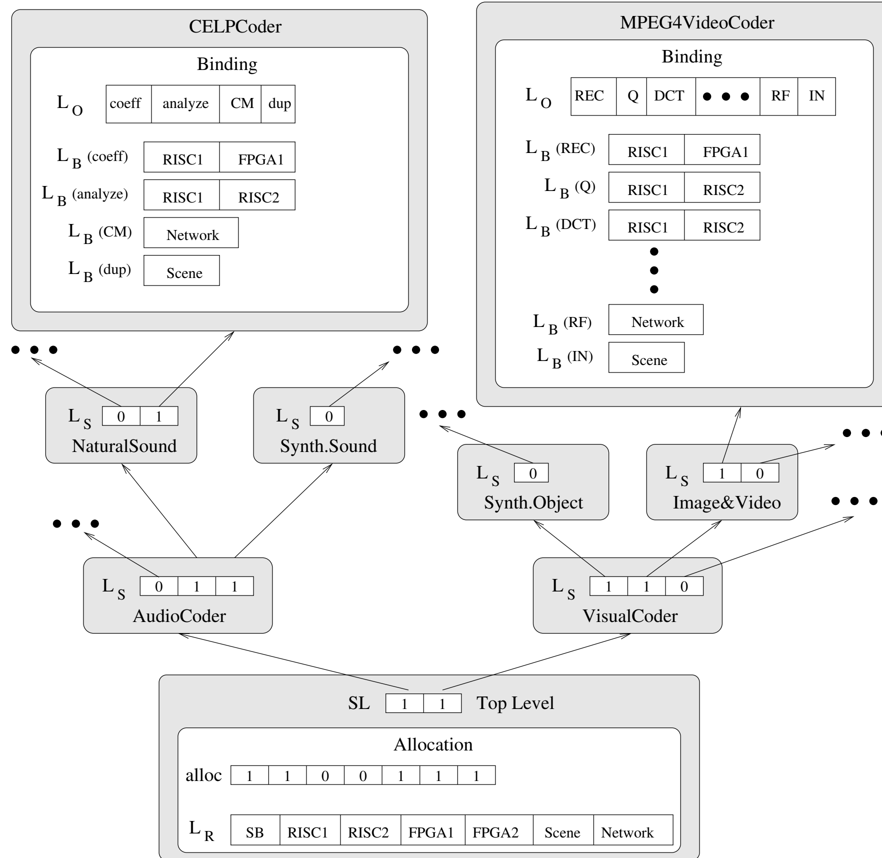

Figure 3.15. Hierarchical chromosome structure for the problem graph given in Figure 3.4 and the architecture graph given in Figure 3.2.

1 *composite mutation*

2 *composite crossover*

These operators are introduced in order to explore design points with different flexibilities.

5.2.1 Composite Mutation. The hierarchical nodes in a hierarchical chromosome directly encode the use of associated subgraphs in the implementation. The composite mutation of a hierarchical chromosome changes a selection bit in a selection list L_S from selected to deselected, or vice versa. As a result, leaves of the chromosome are selected or deselected. Figure 3.16 shows both cases.

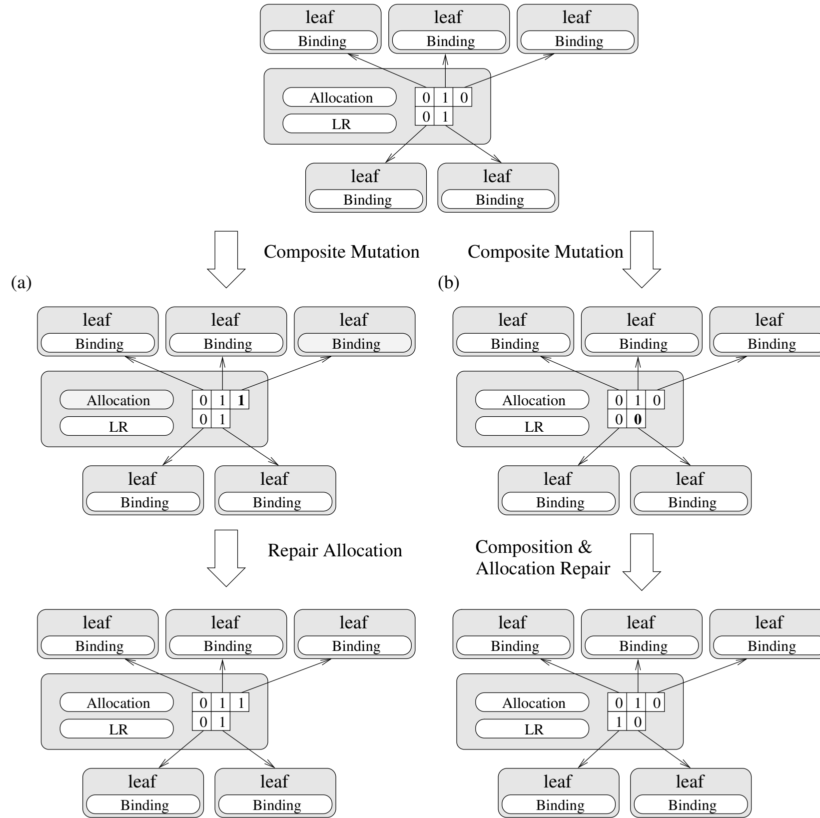

Figure 3.16. Example of composite mutation in hierarchical chromosome (a) selection of a leaf graph (b) deselection of a leaf graph.

The selection of an additional leaf is depicted in Figure 3.16(a). Since the selection of a leaf in the chromosome corresponds to the allocation of a leaf graph in the problem graph, we may produce infeasible results: A newly allocated leaf graph may have an inconsistent binding regarding the global allocation of the chromosome. Perhaps the leaf also contains unmappable operations under the given allocation. Thus in a first step, we have to repair the allocation on the provided information of not yet bound operations of the newly allocated leaf. This has to be repeated until a feasible binding is found for all allocated leaf graphs.

The case of deselection of a leaf is shown in Figure 3.16(b). In this case no leaf graph remains allocated for one of the two hierarchical vertices. Consequently, a repair of the allocation is necessary. Here, we select one associated subnode randomly. This step is called *composition repair*. The allocation of the new selected leaf will be repaired subsequently.

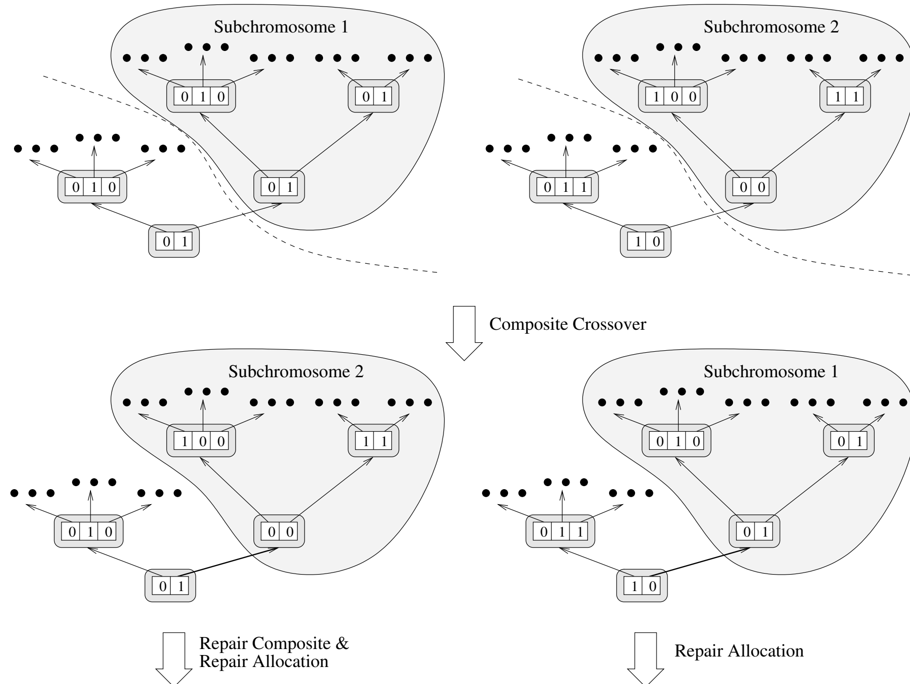

Figure 3.17. Example of composite crossover applied to a hierarchical chromosome resulting in one valid and one invalid composition.

In summary, composite mutation is used in order to explore the design space of different allocations of leaf graphs of the problem graph (flexibility). The second genetic operator, composite crossover, is used for the same purpose, allowing, however larger changes as when using the composite mutation operator only.

5.2.2 Composite Crossover. The second operator is called composite crossover. An example of how composite crossover works is shown in Figure 3.17. Two individuals are cut at the same hierarchical node in the chromosome. After that we interchange these two subgraphs. This results in two new chromosomes as depicted in Figure 3.17. This operation again may invalidate one or both implementations represented by the chromosomes.

Since we store the allocation in the top-level node of an individual, we know the allocation for both resulting chromosomes. Again, this operation could invalidate the leaf graph allocation (at least one subnode has to be selected for each selected hierarchical node in a chromosome, see lower left part in Figure 3.17). Thus, we have to repair the composition of the implementation

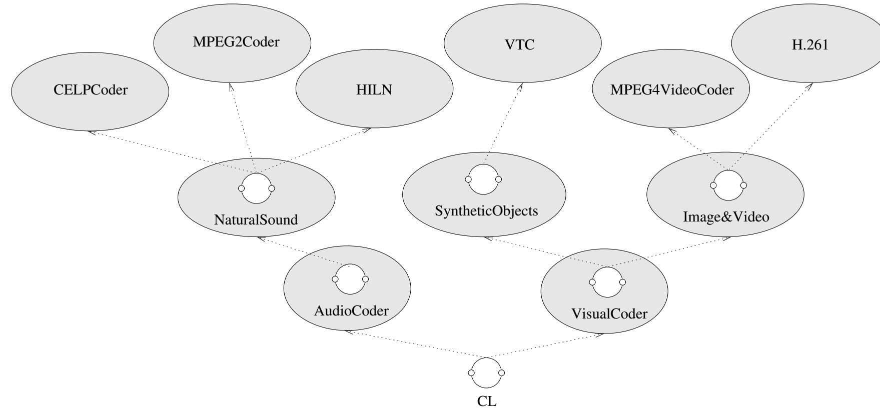

Figure 3.18. Complete functional problem graph specification of the MPEG4 coder.

first. This can be established by repairing the node in the next higher level, i.e., deselecting the inserted node. But this, again, may lead to an invalid leaf graph allocation. The better choice is to repair the inserted node by selecting an appropriate subgraph. In a second step, we have to repair the global allocation as described above.

6. Case Study

This section presents first results obtained by using our two new approaches.

6.1 Example

Figure 3.18 shows the complete functional specification for the MPEG4 coder used throughout the rest of this chapter. There are six leaf graphs, each representing a different coding algorithm. Our goal is to implement at least one of these algorithms with the goal to minimize cost, power, and to maximize the flexibility.

As described in Section 2, we also need the architecture on which we can execute the tasks given by the problem graph. Here, we use the same architecture template for all subgraphs.

The underlying architecture is depicted in Figure 3.19. The architecture consists of three shared buses (SBS, SBM, and SBF), two memory modules (a single and a dual ported memory), two programmable RISC processors, a signal processor (DSP), several predefined hardware modules (namely a block matching module (BMM), a module for performing DCT/IDCT operations (DCTM), an add/subtract module (SAM), a Huffman coder (HC)), and I/O devices (INM and OUTM).

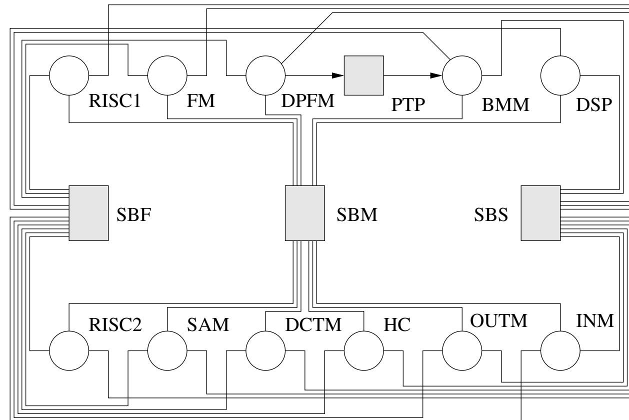

Figure 3.19. Architecture graph for the implementation of the MPEG4 coder.

The RISC processors (RISC1 and RISC2) as well as the signal processor (DSP) are capable of performing any functional operation. However, the DSP executes the operations faster and is more expensive in acquisition. The other hardware modules are dedicated to special groups of operations. For example, the DCTM can only perform the operations discrete cosine transformation and the inverse discrete cosine transformation.

A detailed description of this case study including possible mappings and delays of all modules as well as the results can be found in [18]. The search space for this example consists of more than 2^{200} points.

6.2 Parameters of the Evolutionary Algorithm

In order to apply an Evolutionary Algorithm successfully to a specific optimization problem, several parameters have to be adjusted. Most important are the coding mechanism and the fitness function that have been described in Section 4.2. Here, we briefly outline the selection scheme and recombination mechanism. The selection method should maintain a high diversity in the population, i.e., not only the particular fitness value of an individual is of interest (as in standard selection schemes), but also its "uniqueness". This means that many copies of a good individual should be avoided but different individuals with a good fitness value should be preserved. This is usually achieved by replacing the most "similar" individual out of a randomly chosen crowd of the population by the new individual if it has a better fitness. The "similarity" introduces a new selection criterion and makes a metric necessary to define similarity. Herein, this metric is the number of differently bound functional nodes. The particular selection method used herein is called *restricted tournament selection* [11]. The specific encoding of an individual makes special

crossover and mutation schemes necessary. In particular, for the allocation α uniform crossover is used, that randomly swaps a bit between two parents with a probability of 0.25. For the lists (repair allocation priority lists L_{R}, binding order lists L_O and the binding priority lists $L_{\mathrm{B}}(v)$), order based crossover (also names position-based crossover) is applied (see [5]). Order based crossover ensures that only permutations of the elements in the chromosomes are created, i.e., parts of the list of the parents are combined and repaired such that a legal permutation is obtained. The construction of the individuals makes further repairing methods unnecessary.

For the composite crossover in hierarchical chromosomes, single-point crossover with a probability of 0.25 is chosen. Single-point crossover is performed on the selection list of each single node in the hierarchical chromosome by randomly choosing a crossing site along the list and by exchanging all bits on the right side of the crossing site.

A mutation of an allocation α consists in simply swapping each allocation bit with a probability of 0.2. The mutation operator for the repair allocation lists creates a new permutation of a list by swapping two randomly chosen elements of the list. Composite mutation is done on the selection list of each node of the hierarchical chromosome, and by swapping one element of the selection list with a probability of 0.2.

In our experiments, we have chosen a population size of 300 and the archive size in the SPEA2 method equal to 70.

6.3 Exploration Results

As due to complexity reasons, we do not know the true Pareto-set, we compare the quality sets obtained by each approach against the quality set obtained by combining all these results and taking the Pareto-set of this union of optimal points. This set possesses 38 Pareto-optimal design points. A good measure of comparing two quality sets A and B is then to compute the so-called *coverage* $\mathcal{C}(A, B)$ as defined in [20]:

Definition 3.18 (Coverage) *The* coverage $\mathcal{C}(A, B)$ *of two sets* A *and* B *is a function that maps the ordered pair* (A, B) *to the interval* $[0, 1]$*:*

$$\mathcal{C}(A, B) = \frac{|\{b \in B \mid \exists a \in A : a \succeq b\}|}{|B|}$$

Obviously, a coverage of $\mathcal{C}(A, B) = 1$ corresponds to the fact that all elements in B are weakly dominated by at least one element of A. On the other hand, a coverage of $\mathcal{C}(A, B) = 0$ means that none of the elements in B is weakly dominated by the elements of A.

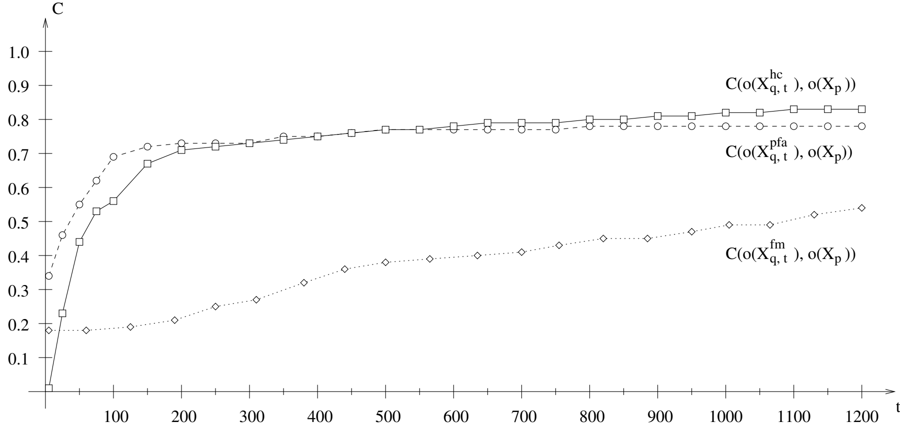

Figure 3.20. Coverage of the Pareto-optimal implementations found after a given number of generations compared to the Pareto-set.

6.3.1 Pareto-Front Arithmetics. When using Pareto-front arithmetics to construct a quality set, we have to

1. Generate leaf graph Pareto-fronts for each leaf graph in Figure 3.18 using one non-hierarchical EA each time.

2. Apply Pareto-front arithmetics to these fronts.

First, we present the results of the design space exploration of each leaf graph. Using the SPEA2 algorithm, all Pareto-points of the leaf graphs were found. For each leaf graph we have calculated 5 different sample runs. The average calculation time τ and number of generations t on a Sun-Blade 100 system equipped with a 450 MHz Sparc-II processor and 2 GByte of main memory are given below:

	CELP	AAC	HILN	VTC	Image	H.261	Total
τ	2 s	19.6 s	600.4 s	2 s	450.4 s	116.6 s	1191.0 s
t	1	3	99.4	1	62.4	18.4	185.2

The times given above show that the exploration of the design space of small leaf graphs may already be very time consuming. Thus, we cannot expect to explore the full flat design space within a reasonable amount of computation time. As described in Section 5.1, we propose Pareto-front arithmetics for fast design space exploration as follows:

With the results for each leaf graph, we can start a quick construction ($< 1s$ for the given example) of our quality set $X^{\mathrm{pfa}}_{\mathrm{q},t}$ after each generation t. Since this computation time is an order of magnitude smaller than the time needed for exploring the top-level design space (as experiments have shown), this approach seems to be a fast method of approximating the Pareto-set X_{p}.

For our example, we obtained a coverage of $\approx 78\%$ of the Pareto-set (see Figure 3.20). This is also the maximum coverage we can expect, since this quality set was constructed from the Pareto-sets of the leaf graphs. Furthermore, using Pareto-front arithmetics, our results converged fast (< 350 generations).

6.3.2 Hierarchical Chromosomes. By using hierarchical chromosomes, we could improve the coverage of the Pareto-set. After $t = 1100$ generations we achieved a coverage of $\mathcal{C}(o(X_{\mathrm{q},t}^{\mathrm{hc}}), o(X_{\mathrm{p}})) \approx 0.83\%$. Again, we averaged the results after 5 different runs.

As Figure 3.20 shows, the hierarchical EA produces only a few Pareto-optimal points at the beginning. This is due to the fact, that now the EA is also responsible for the exploration of allocations in the problem graph.

In contrast to the results when using Pareto-front arithmetics, the hierarchical EA could reach a full coverage of the Pareto-set when run sufficiently long. As we can see in Figure 3.20, the hierarchical EA produces a better quality set as Pareto-front arithmetics already after $t \approx 350$ generations.

6.3.3 Non-Hierarchical EAs. In a last step, we compare our two new approaches against a non-hierarchical approach. In this non-hierarchical exploration algorithm, we explore the design spaces individually for all $2^k - 1$ possible combinations where k is the total number of leaf subgraphs in the problem graph. Therefore, we perform six different exploration runs for each individual leaf subgraph, $\binom{6}{2}$ = 15 runs for combinations that select exactly two leaf subgraphs, etc. All in all, there are $2^6 - 1 = 63$ combinations of leaf subgraphs, where at least one leaf subgraph has to be chosen.[3]

For each of these 63 cases we apply the EA for a certain number of generations for each combination k to obtain the quality set of the different leaf graph selections. Since we use the same number of generations for each combination, we simulate the case were each combination is selected with the same probability. With the given archives $\overline{P}_{t,k}$, we are able to construct the quality set of the top-level design, denoted by $X_{\mathrm{q},t}^{\mathrm{fm}}$, by simply taking the union of all archives $\overline{P}_{t,k}$ of the combinations and calculating the Pareto-optimal points in the union. Figure 3.20 shows the result compared with the Pareto-set of our particular problem.

For our particular problem, we see that both Pareto-front arithmetics as well as hierarchical EAs are superior to the non-hierarchical exploration. By using the flattened model, the coverage of the Pareto-front is $\mathcal{C}(X_{\mathrm{q},1200}^{\mathrm{fm}}, X_{\mathrm{p}}) \approx 52\%$. However, as in the case of the hierarchical chromosomes, it should be possible to find all Pareto-optimal solutions by using this non-hierarchical EA.

7. Conclusions

Two novel approaches, namely *Pareto-front arithmetics* and *Hierarchical Chromosomes* are proposed in this chapter and applied to the problem of synthesis of embedded systems using Evolutionary Algorithms. We propose a hierarchical graph-based model to describe algorithms, sets of architectures, and mapping constraints in order to formalize the task of optimizing an implementation.

Since designing a system to best meet a set of requirements on cost, power, and flexibility implies multi-objective optimization, special Evolutionary Algorithms called MOEAs (Multi-Objective Evolutionary Algorithms) are applied here to the complex task of system synthesis. MOEAs have been found valuable in systems synthesis due to the sophisticated repairing function of invalid individuals. Contrary to the punishment of invalid individuals, this repairing increases the effectiveness of the algorithm by producing many feasible implementations. Also, EAs seem to be good at exploring selection and assignment problems that are dominant in the area of system synthesis.

Nevertheless, system complexity grows steadily leading to giant search spaces demanding new hierarchical strategies in design space exploration. In this chapter, we therefore proposed Pareto-front arithmetics for fast design space exploration that performs arithmetics only on the Pareto-optimal points of subsystems to construct a quality set of the top-level specification. This approach has proven to find a substantial number of Pareto-optimal and additional feasible implementations while reducing the computation time dramatically. A second approach encodes the hierarchical structure in the chromosome of the MOEA. Although being slower in the convergence speed, this approach is able to find better quality sets in the long term.

In the future, we will extend the dimension of the objective space further to include timing analysis and scheduling issues.

Notes

1. The relations $\circ \in \{=, \leq, <, \geq, >\}$ are defined as: $o(i) \circ o(\tilde{i})$ iff $\forall j = 1, \ldots, n : o_j(i) \circ o_j(\tilde{i})$.
2. Without loss of generality, we assume that all objectives are to be minimized in the following.
3. Note that this method is in general not a feasible way to go as the number of EA runs grows exponentially with the number of leaf graphs.

References

[1] Santosh G. Abraham, B. Ramakrishna Rau, and Robert Schreiber. Fast Design Space Exploration Through Validity and Quality Filtering of Subsystem Designs. Technical report, Hewlett Packard, Compiler and Architecture Research, HP Laboratories Palo Alto, July 2000.

[2] J. Axelsson. Architecture Synthesis and Partitioning of Real-Time Systems: A Comparision of Three Heuristic Search Strategies. In *5th International Workshop on Hardware/Software Codesign*, pages 161–166, Braunschweig, 1997. IEEE Computer Society Press.

[3] T. Blickle, J. Teich, and L. Thiele. System-Level Synthesis Using Evolutionary Algorithms. In Rajesh Gupta, editor, *Design Automation for Embedded Systems*, 3, pages 23–62. Kluwer Academic Publishers, Boston, January 1998.

[4] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In *Proc. of the Parallel Problem Solving from Nature VI (PPSN-VI)*, pages 849 – 858, 2000.

[5] Kalyanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley & Sons, 2001.

[6] R.P. Dick and N.K. Jha. MOGAC: A Multiobjective Genetic Algorithm for Hardware-Software Cosynthesis of Distributed Embedded Systems. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 17(10)*, pages 920–935, 1998.

[7] K. Dussa-Zieger. *Model-Based Scheduling and Configuration of Heterogeneous Parallel Systems*. PhD thesis, University of Erlangen-Nürnberg, 1998. Arbeitsberichte des Instituts für Mathematische Maschinen und Datenverarbeitung (IMMD), 31(12).

[8] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu-Search. In *Design Automation for Embedded Systems. Kluwer Academic Publisher, Boston, 2(1)*, pages 5–32, 1997.

[9] R. Ernst, J. Henkel, T. Benner, W. Ye, U. Holtmann, D. Herrmann, and M. Trawny. The COSYMA Environment for Hardware/Software Cosynthesis of Small Embedded Systems. In *Microprocessors and Microsystems 20(3)*, pages 159–166. Elsevier Science B.V., 1996.

[10] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

[11] Georges R. Harik. Finding Multimodal Solutions Using Restricted Tournament Selection. In Larry J. Eshelman, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms (ICGA6)*. Morgan Kaufmann, 1995.

[12] Christian Haubelt, Jürgen Teich, Kai Richter, and Rolf Ernst. Flexibility/Cost-Tradeoffs in Platform-Based Design. In E.F. Deprettere,

J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges*, volume 2268 of *Lecture Notes in Computer Science (LNCS)*, pages 38–56, Berlin, Heidelberg, March 2002. Springer.

[13] J. Henkel and R. Ernst. An Approach to Automated Hardware/Software Partitioning Using a Flexible Granularity that is Driven by High-Level Estimation Techniques. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 9(2)*, pages 273–289, 2001.

[14] Joshua Knowles and David Corne. The Pareto Archived Evolution Strategy: A New Baseline Algorithm for Pareto Multiobjective Optimisation. In *1999 Congress on Evolutionary Computation*, pages 98–105, Piscataway, NJ, 1999. IEEE Service Center.

[15] Vilfredo Pareto. *Cours d'Économie Politique*, volume 1. F. Rouge & Cie., Lausanne, Switzerland, 1896.

[16] Günter Rudolph. On a Multi-Objective Evolutionary Algorithm and Its Convergence to the Pareto Set. In *Proceedings of the 5th IEEE Conference on Evolutionary Computation*, pages 511–516, Piscataway, New Jersey, 1998. IEEE Press.

[17] Günter Rudolph and Alexandru Agapie. Convergence Properties of Some Multi-Objective Evolutionary Algorithms. In *Proc. of the 2000 Congress on Evolutionary Computation*, pages 1010–1016, Piscataway, NJ, 2000. IEEE Service Center.

[18] Jürgen Teich, Christian Haubelt, Sanaz Mostaghim, Frank Slomka, and Ambrish Tyagi. Techniques for Hierarchical Design Space Exploration and their Application on System Synthesis. Technical Report 1/2002, Institute Date, Department of EE and IT, University of Paderborn, Paderborn, Germany, 2002.

[19] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical report, Swiss Federal Institute of Technology (ETH) Zurich, 2001. TIK-Report 103. Department of Electrical Engineering.

[20] Eckart Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Department of Electrical Engineering, Swiss Federal Institute of Technology (ETH) Zurich, December 1999.

[21] Eckart Zitzler and Lothar Thiele. An Evolutionary Algorithm for Multiobjective Optimization: The Strength Pareto Approach. Technical Report 43, Swiss Federal Institute of Technology (ETH) Zurich, Zurich, Switzerland, 1998.