System Design for Flexibility[*]

Christian Haubelt, Jürgen Teich
DATE, University of Paderborn
Paderborn, Germany
{haubelt|teich}@date.upb.de.

Kai Richter, Rolf Ernst
IDA, Technical University of Braunschweig
Braunschweig, Germany
{richter|ernst}@ida.ing.tu-bs.de.

Abstract

With the term flexibility*, we introduce a new design dimension of an embedded system that quantitatively characterizes its feasibility in implementing not only one, but possibly several alternative behaviors. This is important when designing systems that may adopt their behavior during operation, e.g., due to new environmental conditions, or when dimensioning a platform-based system that must implement a set of different behaviors. A* hierarchical graph model *is introduced that allows to model flexibility and cost of a system formally. Based on this model, an efficient exploration algorithm to find the* optimal flexibility/cost-tradeoff-curve *of a system using the example of the design of a family of Set-Top boxes is proposed.*

1. Introduction

Designing a system to best meet a set of requirements on cost, speed, power, etc. for a given, single application is challenging, but has been formalized already by means of graph-based allocation and binding problems such as [2]. Such graphical mapping models found acceptance also in commercial systems such as [3].

In areas such as platform-based design, however, a system should be dimensioned such that it is able to implement not only one particular application optimally, but instead a complete set of different applications or variants of a certain application. Hence, the question here becomes to find a tradeoff between the flexibility of the architecture that is able to implement several alternative behaviors and its cost.

Another scenario where flexibility is necessary is in systems that may react to changes of the environment during operation (adaptive systems). There also, it is necessary or desired to implement different behaviors with the price of additional cost.

Yet there exist no approaches to the best of our knowledge that quantitatively tradeoff the price one has to pay in terms of additional memory, hardware, network, etc. for the flexibility one gains when dimensioning a system such that it is able to implement multiple behaviors. Pop et al. [10] perform mapping and scheduling such that there is a high probability that new functionality can easily be mapped on the resulted system. Nevertheless, this basically similar idea of flexibility can not guarantee that future applications do not interfere with the already running functionality.

Here, we introduce the notion of *flexibility* as a tentative to quantitatively describe the functional richness that the system under design is able to implement (Section 3). In order to describe a set of applications, a hierarchical specification is useful such as [4] and [11]. Here, we introduce a *hierarchical graph-model* for describing alternatives of the behavior of a system. The same idea may be used in order to describe reconfigurable architectures on the implementation side, i.e., systems that change their structure over time.

With this model, we are then able to define the problem of dimensioning a system that is able to dynamically switch its behavior and/or structure at run-time. Basically, this problem extends previous approaches such as [2] to reconfigurable, platform-based systems that implement time-dependent functionality.

Finally, an efficient exploration algorithm for exploring the flexibility/cost-tradeoff-curve of a system under design is presented that efficiently prunes solutions that are not optimal with respect to both criteria (Section 4). The example of a flexible video Set-Top box is used as the guiding example throughout the paper.

2. Hierarchical Specification Model

Each embedded system is developed to cover a certain range of functionality. This coverage depends on the different types of tasks as well as on the scope every task is able to process. A specification of such an embedded system is depicted in Fig. 1.

[*]This work was supported by the German Science Foundation (DFG), SPP 1040.

The specification shows interacting processes of a digital television decoder. There are four top-level processes, $\mathrm{P_A}$ to handle the authentification process, $\mathrm{P_C}$ to control channel selection, frequency adjustment, etc., $\mathrm{I_D}$ which performs decryption, and $\mathrm{I_U}$ for uncompression. Here, the uncompression process requires input data from the decryption process. Furthermore, the controller and authentification process are well known and are most likely to be implemented equally in each decoder.

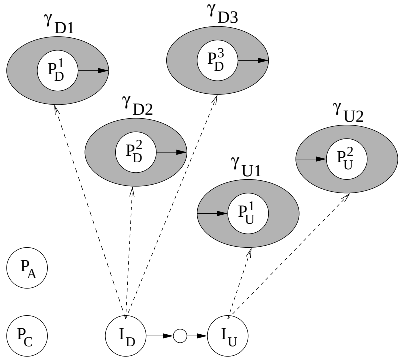


Figure 1. Specification of a Digital TV Decoder

The main difference between TV decoders is made up by the implemented combinations of decryption and uncompression algorithms. As shown in Fig. 1, we use hierarchical refinement to capture all alternative realizations. There are three decryption and two uncompression processes used in this decoder. Obviously, if we implement even more of these refinements, our decoder will support a greater number of TV stations. Consequently, the decoder possesses an increased flexibility.

Before defining a system's flexibility formally, we have to introduce a specification model that is able to express flexibility. As shown in Fig. 1, our specification model is based on the concept of hierarchical graphs.

Definition 1 (hierarchical graph) *A* hierarchical graph G *is a tuple* $G = (V, E, \Psi, \Gamma)$*, with* V *and* E *being the set of vertices and edges, respectively.* Ψ *denotes a set of so called* interfaces*, i.e., the set of hierarchical vertices, which are refined by the use of alternative* clusters $\gamma \in \Gamma$*, i.e., subgraphs.*

Figure 1 shows a digital TV decoder as a hierarchical graph with its top-level graph depicted at the bottom. The top-level graph consists of two non-hierarchical vertices, $V = \{\mathrm{P_A}, \mathrm{P_C}\}$ and two interfaces ($\Psi = \{\mathrm{I_D}, \mathrm{I_U}\}$). The decryption interface $\mathrm{I_D}$ itself can be refined by three clusters γ_{D1}, γ_{D2}, and γ_{D3}, where each cluster represents an alternative refinement of $\mathrm{I_D}$. The set of clusters is given by $\Gamma = \{\gamma_{\mathrm{D1}}, \gamma_{\mathrm{D2}}, \gamma_{\mathrm{D3}}, \gamma_{\mathrm{U1}}, \gamma_{\mathrm{U2}}\}$.

Clusters (subgraphs) are defined in analogy to hierarchical graphs. Since the in-degree and out-degree of an interface is not limited, we need the notion of *ports*. Interfaces are connected with vertices or other interfaces via ports. These ports are used to embed clusters into a given interface. In the following, this process is called *port mapping*.

All clusters associated with the interface ψ represent *alternative refinements* of ψ. The process of *cluster-selection* associated with each interface ψ determines exactly one cluster to implement ψ at each instant of time. In order to avoid a loss of generality, we do not restrict cluster-selection to system start-up. Thus, reconfigurable and adaptive systems may be modeled via time-dependent switching of clusters.

The set of leaves of a hierarchical graph G is defined by the recursive equation:[1]

$$V_\mathrm{l}(G) = G.\mathrm{V} \cup \bigcup_{\psi \in G.\Psi} \bigcup_{\gamma \in \psi.\Gamma} V_\mathrm{l}(\gamma) \qquad (1)$$

As defined by Equation (1), the set of leaves $V_\mathrm{l}(G)$ of graph G shown in Figure 1 computes to $V_\mathrm{l}(G) = \{\mathrm{P_A}, \mathrm{P_C}\} \cup \{\gamma_{\mathrm{D1}}.\mathrm{P}_\mathrm{D}^1, \gamma_{\mathrm{D2}}.\mathrm{P}_\mathrm{D}^2, \gamma_{\mathrm{D3}}.\mathrm{P}_\mathrm{D}^3\} \cup \{\gamma_{\mathrm{U1}}.\mathrm{P}_\mathrm{U}^1, \gamma_{\mathrm{U2}}.\mathrm{P}_\mathrm{U}^2\}$.

So far, we only considered the behavioral part of the specification. For implementation, we also require information about the possible structure of our system, i.e., the underlying architecture. This leads to a graphical model for embedded system specification, the so called *specification graph* $G_\mathrm{S} = (G_\mathrm{P}, G_\mathrm{A}, E_\mathrm{M})$. It mainly consists of three components: a *problem graph*, an *architecture graph*, and *user-defined mapping* edges (see also [2]). The respective graphs G_P and G_A are based on the concept of hierarchical graphs as defined in Def. 1.

Problem Graph. The *problem graph* G_P is a directed hierarchical graph $(V_\mathrm{P}, E_\mathrm{P}, \Psi_\mathrm{P}, \Gamma_\mathrm{P})$ for modeling the required system's behavior (see Fig. 1). Vertices $v \in V_\mathrm{P}$ and interfaces $\psi \in \Psi_\mathrm{P}$ represent processes or communication operations at system-level. The edges $e \in E_\mathrm{P}$ model dependence relations, i.e., define a partial ordering among the operations. The clusters $\gamma \in \Gamma_\mathrm{P}$ are possible substitutions for the interfaces $\psi \in \Psi_\mathrm{P}$.

Architecture Graph. The class of possible architectures is modeled by a directed hierarchical graph $G_\mathrm{A} = (V_\mathrm{A}, E_\mathrm{A}, \Psi_\mathrm{A}, \Gamma_\mathrm{A})$, called *architecture graph*. Functional or communication resources are represented by vertices $v \in V_\mathrm{A}$ and interfaces $\psi \in \Psi_\mathrm{A}$, interconnections are specified by the edges $e \in E_\mathrm{A}$. Again, the clusters $\gamma \in \Gamma_\mathrm{A}$ represent potential implementations of the associated interfaces. All the resources are viewed as potentially allocatable components.

Mapping Edges. *Mapping edges* $e \in E_\mathrm{M}$ indicate user-defined constraints in the form of a "can be implemented by"-relation. These edges link leaves $V_\mathrm{l}(G_\mathrm{P})$ of the problem graph G_P with leaves $V_\mathrm{l}(G_\mathrm{A})$ of the architecture graph G_A.

[1] The $G.V$ notation is used as decomposition operation, e.g., to access the set of vertices V inside the graph G.

Additional parameters, like priorities, power consumption, latencies, etc., which are used for formulating implementational and functional constraints are annotated to the components of G_{S}. For simplicity, a specification graph can also be represented only by its vertices and edges: $G_{\mathrm{S}} = (V_{\mathrm{S}}, E_{\mathrm{S}})$. The set of vertices V_{S} covers all non-hierarchical vertices, interfaces, and clusters contained in the problem or architecture graph. The set of edges E_{S} consists of all edges and port mappings in the specification graph.

An example of a specification graph is shown in Figure 2. Again, the problem graph specifies the behavior of our digital TV decoder. The architecture graph is depicted on the right. It is composed of a μ-Controller (μP), an ASIC (A), and an FPGA. There are two busses C_1 and C_2 to handle the communication between the μ-Controller and FPGA and ASIC, respectively. Figure 2 also shows the allocation costs for each resource in the architecture graph.

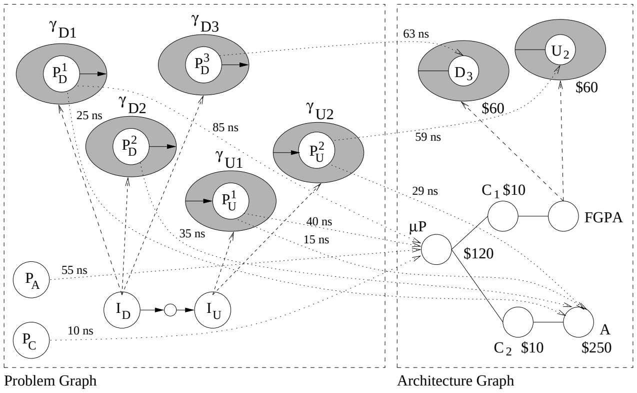


Figure 2. Hierarchical Specification Graph

The mapping edges (dotted edges in Fig. 2) outline the possible bindings of processes of the problem graph to resources of the architecture graph. The latencies to execute a given process on a specific resource are annotated to the respective mapping edges. For example, the uncompression process $\mathrm{P}_{\mathrm{U}}^{1}$ is executable on resource μP with a latency of 40 ns or on resource A with a latency of 15 ns.

As shown in Figure 2, the hierarchical specification graph permits modeling of adaptive systems by interchanging clusters in the problem graph. In our example, we have to select a certain decryption and uncompression process to match the requirements imposed by the TV station. Generally, an adaptive system responds to environmental changes by selecting clusters according to the requirements of input/output data at runtime. Therefore, clusters with various parameters or perhaps totally different functionality are activated in the problem graph. On the other hand, interchanging clusters in the architecture graph modifies the structure of the system. If this cluster-selection is performed at runtime, the architecture model characterizes reconfigurable hardware. For example, in order to execute process $\mathrm{P}_{\mathrm{D}}^{3}$, we have to configure the FPGA with the respective design D_3 (see Figure 2).

In order to specify an *implementation*, i.e., a concrete mapping, Blickle et al. [2] introduce the concept of *activation* of vertices and edges. The activation of a specification graph's vertex or edge describes its use in the implementation. Since we use hierarchical graphs, we have to define *hierarchical timed activation* or, for short, *hierarchical activation*, where the *hierarchical activation* of a specification graph is a boolean function that assigns to each edge and to each vertex the value 1 (activated) or 0 (not activated) at a given time $t \in T(= \mathbb{R})$.

Hierarchical activation should support synthesis in such a way that no infeasible implementations are caused by the following rules. Due to space limitations, we summarize these implications informally. For full description, see [6].

1. The activation of an interface at time t implies the activation of exactly one associated cluster at the same time.
2. The activation of a cluster γ at time t activates all embedded vertices and edges in γ.
3. Each activated edge $e \in E_{\mathrm{S}}$ has to start and end at an activated vertex. This must hold for all times $t \in T$.
4. Due to (perhaps implied) timing constraints, the activation of all top-level vertices and interfaces in the problem graph G_{P} is required.

For a given selection of clusters, the hierarchical model can be flattened. With the formalism of hierarchical activation rules, we are able to determine the overall activation of the specification graph. The result is a non-hierarchical specification.

With the definition of hierarchical activation, we are able to formally define the term *implementation*, where a feasible implementation consists of a feasible *allocation* and a corresponding feasible *binding*.

Definition 2 (timed allocation) *A* timed allocation $\alpha(t)$ *of a specification graph* G_{S} *is the subset of all activated vertices and edges of the problem and architecture graph at time* t*.*

Definition 3 (timed binding) *A* timed binding $\beta(t)$ *is the subset of all activated mapping edges at time* t*.*

In order to restrict the combinatorial search space, it would be useful to determine the set of feasible timed allocation and feasible timed binding. First, we consider the feasibility of a binding for a given specification graph G_{S} and a given timed allocation $\alpha(t)$. A *feasible timed binding* $\beta(t)$ satisfies the following requirements:

1. Each activated edge $e \in \beta(t)$ starts and ends at a vertex, activated at time t.
2. For each activated leaf $v \in \{V_{\mathrm{l}}(G_{\mathrm{P}}) \cap \alpha(t)\}$ of the problem graph G_{P}, exactly one outgoing edge $E \in E_{\mathrm{M}}$ is activated at time t.

3. For each activated edge $e = (v_i, v_j) \in E_{\mathrm{P}} \cap \alpha(t)$:
   - either both operations are mapped onto the same vertex, i.e.,
   - or there exists an activated edge $\widetilde{e} = (\widetilde{v}_i, \widetilde{v}_j) \in \{E_{\mathrm{A}} \cap \alpha(t)\}$ to handle the communication associated with edge e, i.e.,

Now, we are able to define the feasibility of a given timed allocation. A *feasible timed allocation* is a timed allocation $\alpha(t)$ that allows at least one feasible timed binding $\beta(t)$ for all times t.

In Figure 2, an infeasible binding would be caused by binding decryption process $\mathrm{P}_{\mathrm{D}}^{2}$ onto the ASIC A and the uncompression process $\mathrm{P}_{\mathrm{U}}^{1}$ onto the FPGA. Since no bus connects the ASIC and the FPGA, there is no way to establish the communication between these processes.

Note that our hierarchical model introduced here extends the model of [2] by two important features:

a) hierarchical graphs allow to model alternatives

b) time-variant allocations and bindings.

These major extensions are necessary to model flexibility (reconfigurability) of the behavior (architecture).

So far, we have not accounted for system performance. Whether or not the implementation meets the application's performance requirements in terms of throughput (e. g. frames per second) and latencies, depends on the existence of a *feasible schedule*. Although it is possible to schedule any feasible implementation as defined above, the resulting schedule may fail performance requirements. Such scheduling or performance analysis is complex, especially for distributed systems, and is not the scope of this paper. Thus, we do not include a complete analysis in the exploration in Section 4. Rather, we quickly estimate the processor utilization and use the 69% limit as defined in [7] to accept or reject implementations due to performance reasons. The approach presented in [10] may be used for scheduling of specifications with data-independent subgraphs.

3. Definition of Flexibility

With the hierarchical specification model described above, we are able to quantify the amount of implemented functionality. Subsequently, we denote this objective *flexibility*. The basic idea, as stated here, is to enumerate the possible interchanges of implementing clusters in the whole system's problem graph. For example, the flexibility of a trivial system with just one activated interface directly increases with the number of activatable clusters.

The key concepts of *flexibility* are as follows:

- Since each cluster represents an alternative for the same functionality, we know that implementing more clusters for a given interface increases system flexibility in the sense that the system may switch at runtime to select a different cluster.
- A cluster itself can contain interfaces, which can be implemented with different degrees of flexibility.
- Although flexibility depends on the implementation, we neglect the impact of the underlying architecture on flexibility, e.g., we do not distinguish whether the flexibility of a system is obtained by the use of either reconfigurable or dedicated hardware components.

With these assumptions, we can define flexibility as:

Definition 4 (flexibility) *The* flexibility f_{impl} *of a given cluster γ is expressed as:*

$$f_{\mathrm{impl}}(\gamma) = a^{+}(\gamma) \cdot \begin{cases} \left[\sum_{\psi \in \gamma.\Psi} \sum_{\widehat{\gamma} \in \psi.\Gamma} f_{\mathrm{impl}}(\widehat{\gamma})\right] \\ -(|\gamma.\Psi| - 1) \;\; \textit{for} \;\; \gamma.\Psi \neq \emptyset \\ 1 \qquad\qquad\qquad\qquad \textit{otherwise} \end{cases}$$

where the term $a^{+}(\gamma)$ describes the activation of the cluster γ in the future. If cluster γ will be selected at any time in the future, $a^{+}(\gamma) = 1$, otherwise 0, meaning it will not be implemented at all.

In other words: The flexibility of a cluster γ, if ever activated, is calculated by the sum of all its interfaces' flexibilities minus the number of its interfaces less 1, and 1 if there is no interface in the given cluster. The flexibility of an interface is the sum of flexibilities of all its associated clusters. The flexibility of a never activated cluster is 0.

For example, consider the problem graph G_{P} shown in Figure 3. This graph is an extension of the TV decoder example in Figure 1. Here, our goal is to design a Set-Top box family which supports multiple applications. Besides the already known digital TV decoder, there are two more possible applications:

1. An Internet browser, consisting of a controller process $\mathrm{P}_{\mathrm{C}}^{\mathrm{I}}$, parser process P_{P} for parsing HTML pages and a formatter process P_{F} for formatting the output.
2. A game console, modeled by a controller process $\mathrm{P}_{\mathrm{C}}^{\mathrm{G}}$, the game's core interface I_{G}, and the graphics accelerator P_{D}. The game's core interface can be refined by three different game classes denoted $\mathrm{P}_{\mathrm{G}}^{1}$, $\mathrm{P}_{\mathrm{G}}^{2}$, and $\mathrm{P}_{\mathrm{G}}^{3}$ in Fig. 3. Since the output is constrained to a minimal frame period and the graphic accelerator depends on data produced by the game core process, also the game's core process has to obey some timing constraints.

The flexibility $f(G_{\mathrm{P}})$ of this problem graph is computed as follows:

$$\begin{aligned} f(G_{\mathrm{P}}) \;\; &= \;\; a^{+}(G_{\mathrm{P}}) \cdot [f(\gamma_{\mathrm{I}}) + f(\gamma_{\mathrm{G}}) + f(\gamma_{\mathrm{D}})] \\ &= \;\; a^{+}(G_{\mathrm{P}}) \cdot \big[a^{+}(\gamma_{\mathrm{I}}) + a^{+}(\gamma_{\mathrm{G}}) \cdot \big[a^{+}(\gamma_{\mathrm{G1}}) + \\ &\qquad a^{+}(\gamma_{\mathrm{G2}}) + a^{+}(\gamma_{\mathrm{G3}})\big] + a^{+}(\gamma_{\mathrm{D}}) \cdot \\ &\qquad \big[a^{+}(\gamma_{\mathrm{D1}}) + a^{+}(\gamma_{\mathrm{D2}}) + a^{+}(\gamma_{\mathrm{D3}}) + \\ &\qquad a^{+}(\gamma_{\mathrm{U1}}) + a^{+}(\gamma_{\mathrm{U2}}) - 1\big]\big] \end{aligned}$$

Based on this equation, the system's flexibility is obtained by specifying the utilization of each cluster γ in the future, denoted by $a^{+}(\gamma)$. If all clusters can be activated

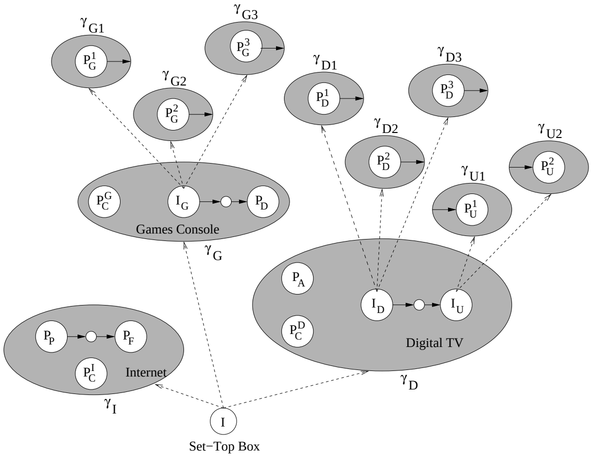


Figure 3. Example for System's Flexibility

in future implementations, system's flexibility calculates to $f(G_\mathrm{P}) = 8$. This is also the maximal flexibility. If, e.g., cluster γ_G is not used in future implementations the flexibility will decrease to $f(G_\mathrm{P}) = 5$.[2] For the sake of simplicity, we have omitted the architecture graph and the mapping edges. Obviously, a cluster only contributes to the total flexibility if it is bindable as described above.

4. Design Space Exploration

Because of the accepted use of tools on lower design levels of abstraction, exploration becomes the next step in order to prevent under- or over-designing a system. Typically, a system has to meet many constraints and should optimize many different design objectives and constraints simultaneously such as execution time, cost, area, power consumption, weight, etc.

A single solution that optimizes all objectives simultaneously is very unlikely to exist. Instead, it should be possible to first explore different optimal solutions or approximations, and subsequently select and refine one of those solutions.

In this paper, we consider the two objectives *flexibility* $f_\mathrm{impl}(\alpha(t))$, as described in Section 4, and *cost* $c_\mathrm{impl}(\alpha(t))$. Here we use the so-called *allocation cost model* $c_\mathrm{impl}(\alpha(t))$, where $c_\mathrm{impl}(\alpha(t))$ is the sum of all realization costs of resources in the allocation $\alpha(t)$.

Figure 4 shows a typical tradeoff-curve between cost and the reciprocal value of flexibility. As already mentioned we are concerned with a MOP (Multiobjective Optimization Problem). Our MOP consists of two objective functions $c_\mathrm{impl}(\alpha(t))$ and $\frac{1}{f_\mathrm{impl}(\alpha(t))}$, where the parameter $\alpha(t)$ is the decision vector. The optimization goal is to minimize $c_\mathrm{impl}(\alpha(t))$ and $\frac{1}{f_\mathrm{impl}(\alpha(t))}$ simultaneously, i.e., to maximize system's flexibility for minimal cost implementations.

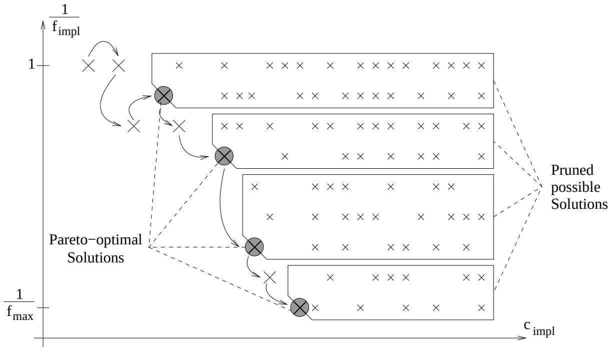


Figure 4. Cost-Flexibility-Design Space

A formal definition of MOPs can be found in [12].

Figure 4 shows four Pareto-optimal design points. A design point is said to be Pareto-optimal iff there is no other design point that is better in all objectives (see also [9]). The goal of design space exploration is to find *all* Pareto-optimal design points that in addition fulfill all timing requirements. The points in Figure 4 represent possible solutions, where not every solution has to be feasible in the sense of a feasible binding and feasible allocation, and not every feasible solution has to meet the timing requirements. If we have found a Pareto-optimal solution x that meets all requirements, the class of all design points dominated by x can be pruned. This is shown in Figure 4 by boxes. In the following, we will introduce an algorithm for efficiently exploring flexibility/cost-tradeoff-curves.

Figure 4 shows an example distribution of design points. At this stage, we do not distinguish between feasible and non-feasible solutions. Our objective is to find all Pareto-optimal solutions that meet all timing constraints. The problem is, that the set of possible implementations is unknown. Since binding is a NP-complete problem (see [2]), an exhaustive search approach (there are $2^{|V_\mathrm{S}|}$ possible solutions) seems not to be a viable solution.

To avoid superfluous computation of non-Pareto-optimal solution, we propose methods for search space reduction:

1. **Possible Resource Allocations**. A *possible resource allocation* is a partial allocation of resources in the architecture graph which allows the implementation of *at least one* feasible problem graph activation by neglecting the feasibility of binding first. Usually, we have to investigate all $2^{|V_\mathrm{S}|}$ design points. But only the elements covering a possible resource allocation represent meaningful activations such that at least a required minimum of problem graph vertices is bindable.

2. **Flexibility Estimation**. With the possible resource allocations we are able to sort the remaining design points by increasing costs. If we inspect the elements of this sorted list by increasing costs, a new calculated solution is Pareto-optimal, iff it possesses a greater flexibility than each solution that has been already implemented.

[2] Note that more sophisticated flexibility calculations are possible, e.g., by using weighted sums in Def. 4.

As shown in our case study (see Section 5), by using these two techniques, we dramatically reduce the invocations of the solver for the NP-complete binding problem.

With our approach of hierarchical specification and activations, we are able to first determine the set of possible resource allocations: For each vertex v_i inside a given cluster γ_j, we determine the set R_{ij} of reachable resources. A resource r is reachable if a mapping edge between v_i and r exists. Derived from the hierarchical activation rules, only leaves $v \in G_{\mathrm{A}}.V$ of the top-level architecture graph or whole clusters of the architecture graph are considered. Next, we set up the outer conjunction R_j of all power sets $2^{R_{ij}}$. Consequently, the set R_j describes *all* combinations of resource activations for implementing the non-hierarchical vertices $v \in \gamma_j.V$ of cluster γ_j by ignoring the feasibility of binding.

Finally, we have to inspect all hierarchical components $\gamma_j.\Psi$ of cluster γ_j. Since all clusters associated with an interface $\psi \in \gamma_j.\Psi$ represent alternative refinements of ψ, we compute the union of possible resource allocations for the associated clusters. For example, the set $\mathcal{A}$ of possible resource allocations for the specification given in Figure 2 computes to:

$$\begin{aligned}\mathcal{A} \;=\; & \{\mu\mathrm{P}, \mu\mathrm{PC}_1, \mu\mathrm{PC}_2, \mu\mathrm{PC}_1\mathrm{C}_2, \mu\mathrm{PD}_3, \mu\mathrm{PU}_2,\\ & \mu\mathrm{PC}_1\mathrm{D}_3, \mu\mathrm{PC}_2\mathrm{D}_3, \mu\mathrm{PC}_1\mathrm{U}_2, \mu\mathrm{PC}_2\mathrm{U}_2,\\ & \mu\mathrm{PC}_1\mathrm{C}_2\mathrm{D}_3, \ldots, \mu\mathrm{PC}_1\mathrm{C}_2\mathrm{D}_1\mathrm{U}_2\mathrm{A}\}\end{aligned}$$

Now, the elements of the set of possible resource allocations are inspected in order of increasing allocation costs c_{impl} (see Fig. 4). For every possible resource allocation, we remove all resources that are not activated from the architecture graph. By removing these elements, also mapping edges are removed from the specification graph. Next, we delete all vertices in the problem graph with no incident mapping edge. This results in a reduced specification graph.

With Def. 4, the maximal flexibility of this specification can be calculated. Since we explore flexibility/cost-objective-space by increasing costs (see Figure 4), we are only interested in design points with a greater flexibility than already implemented! With the known maximal implemented flexibility, we therefore may skip specifications with a lower implementable flexibility. For specifications with greater expected flexibility, we try to construct a feasible implementation next.

Generally, more than one activatable cluster for a problem graph's interface remains in the specification graph. Consequently, we have to identify so-called *elementary cluster-activations*, which are defined as follows. Let Γ_{act} denote the set of activatable clusters. An elementary cluster-activation ecs is a set $ecs = \{\gamma_i \mid \gamma_i \in \Gamma_{\mathrm{act}}\}$, where exactly one cluster is selected per activated interface. Since every activatable cluster has to be part of the implementation to obtain the expected flexibility, we have to determine a coverage [5] of Γ_{act} by elementary cluster-activations.

As example consider Figure 2. For a given resource allocation $\mu\mathrm{PC}_2\mathrm{A}$, the clusters $\gamma_{\mathrm{D1}}, \gamma_{\mathrm{D2}}, \gamma_{\mathrm{U1}}$, and γ_{U2} are activatable. One coverage of this set is given by the elementary cluster-activations $\{\gamma_{\mathrm{D}_2}\gamma_{\mathrm{U1}}\}$ and $\{\gamma_{\mathrm{D1}}\gamma_{\mathrm{U}_2}\}$.

Given an elementary cluster-activation, we can select these clusters for implementation. Furthermore, we must determine valid cluster activations in the architecture graph, so that every elementary cluster-activation can be bound to a non-ambiguous architecture, i.e., there is exactly one activated cluster for every activated interface in the architecture graph.

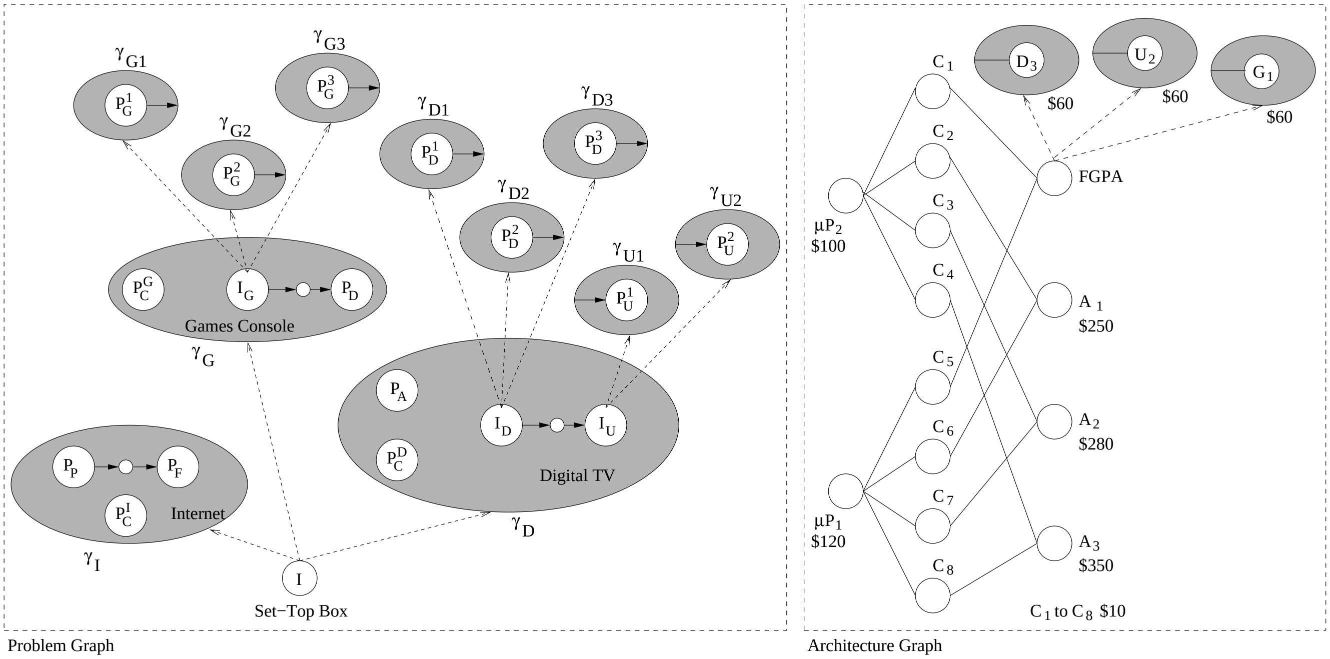
Finally, we validate all timing constraints that are imposed on our implementation. Here, we use a statistical analysis method to check for fulfillment. With these basic ideas of pruning the search space, we formulate our exploration algorithm based on a branch-and-bound strategy [5, 8]. For the sake of clarity, we omit details for calculating a coverage of activatable problem graph clusters or successive flexibility estimation, etc. The following code should be self-explanatory with the previous comments.

```
EXPLORE
  IN:     specification graph $G_{\mathrm{S}}$
  OUT:    Pareto-optimal set $\mathcal{O}$
  BEGIN
    $f_{\mathrm{cur}} = 0$
    $\mathcal{A}$ = $G_{\mathrm{S}}$.possibleResourceAllocations()
    $f_{\max}$ = $G_{\mathrm{S}}$.computeMaximumFlexibility()
    FOR each candidate $a \in \mathcal{A}$ DO
      $f = a$.computeMaximumFlexibility()
      WHILE $f < f_{\mathrm{cur}}$ THEN
        $\alpha = G_{\mathrm{S}}$.computeAllocation(a)
        $\beta = G_{\mathrm{S}}$.computeBinding($\alpha$)
        $i$ = new Implementation($\alpha$, $\beta$)
        IF $i$.isFeasibleImplementation() THEN
          IF $i$.meetsAllConstraints() THEN
            IF $i$.flexibility() $> f_{\mathrm{cur}}$ THEN
              $\mathcal{O}$ = $\mathcal{O} \cup i$
              $f_{\mathrm{cur}}$ = $i$.flexibility()
            ENDIF
          ENDIF
        ENDIF
      ENDWHILE
    ENDFOR
  END
```

In the worst case, this algorithm is not better than an exhaustive search algorithm. But, a typical search space with 10^5-10^{12} design points can be reduced by the EXPLORE-algorithm to a few 10^3-10^4 possible resource allocations. Since we only try to implement design points with a greater expected flexibility than the already implemented flexibility, again, only a small fraction of these point has to be taken into account, typically less than 100.

5. Case Study

In our case study we investigate the specification of our Set-Top box depicted in Figure 5. Again, we increased the complexity of our example. The architecture graph is



Problem Graph

Architecture Graph

Figure 5. Specification of a Set-Top Box

now composed of two processors ($\mu\mathrm{P}_1$ and $\mu\mathrm{P}_2$), three ASIC (A_1 to A_3), and an FPGA. The ASICs are used to improve performance for the decryption, uncompression, game's core, and graphic acceleration processes. The FPGA can also be used as coprocessor for the third decryption, the second uncompression, or the first game core class. The allocation costs of each component are annotated in Fig. 5.

Table 1. Possible Mappings in Figure 5

Process	$\mu\mathrm{P}_1$	$\mu\mathrm{P}_2$	A_1	A_2	A_3	D3	U2	G1
$\mathrm{P}_\mathrm{C}^\mathrm{I}$	10	12	-	-	-	-	-	-
P_P	15	19	-	-	-	-	-	-
P_F	50	75	-	-	-	-	-	-
$\mathrm{P}_\mathrm{C}^\mathrm{G}$	25	27	-	-	-	-	-	-
P_G^1	75	95	15	15	15	-	-	20
P_G^2	-	-	25	22	22	-	-	-
P_G^3	-	-	50	45	35	-	-	-
P_D	70	90	30	30	25	-	-	-
$\mathrm{P}_\mathrm{C}^\mathrm{D}$	10	10	-	-	-	-	-	-
P_A	55	60	-	-	-	-	-	-
P_D^1	85	95	25	22	22	-	-	-
P_D^2	-	-	35	33	32	-	-	-
P_D^3	-	-	-	-	-	63	-	-
P_U^1	40	45	15	12	10	-	-	-
P_U^2	-	-	29	27	22	-	59	-

In Figure 5, we have omitted the mapping edges. Possible mappings and respective core execution times are given in ns as shown in Table 1. Furthermore, we assume that all communications can be performed on every resource. No latencies for external communications are taken into account. Timing constraints for the game console and digital TV are given by the minimal periods of the output processes (P_D, P_U^1, and P_U^2). P_D has to be executed every 240 ns. The output for the digital TV box is less restrictive: P_U^1 and P_U^2 should be executed at least all 300 ns if activated.

As described above, our algorithm starts with the determination of the set of all possible resource activations. Here, elements that are obviously not Pareto-optimal or no feasible implementations are left out, e. g., all combinations of a single functional component and an arbitrary number of communication resources. The beginning of the ordered subset $\mathcal{A}$ of possible resource allocations is given by:

$$\begin{aligned}\mathcal{A} \quad = \quad & \{\mu\mathrm{P}_2, \mu\mathrm{P}_1, \mu\mathrm{P}_2\mathrm{D}_3\mathrm{C}_1, \mu\mathrm{P}_2\mathrm{U}_2\mathrm{C}_1, \mu\mathrm{P}_2\mathrm{G}_1\mathrm{C}_1,\\ & \mu\mathrm{P}_1\mathrm{D}_3\mathrm{C}_5, \mu\mathrm{P}_1\mathrm{U}_2\mathrm{C}_5, \mu\mathrm{P}_1\mathrm{G}_1\mathrm{C}_5, \mu\mathrm{P}_2\mathrm{D}_3\mathrm{U}_2\mathrm{C}_1,\\ & \mu\mathrm{P}_2\mathrm{D}_3\mathrm{G}_1\mathrm{C}_1, \mu\mathrm{P}_2\mathrm{U}_2\mathrm{G}_1\mathrm{C}_1, \mu\mathrm{P}_1\mathrm{D}_3\mathrm{U}_2\mathrm{C}_5,\\ & \mu\mathrm{P}_1\mathrm{D}_3\mathrm{G}_1\mathrm{C}_5, \mu\mathrm{P}_1\mathrm{U}_2\mathrm{G}_1\mathrm{C}_5, \mu\mathrm{P}_1\mu\mathrm{P}_2, \ldots\}\end{aligned}$$

Next, we determine all elementary cluster activations that can be activated under the given resource allocation. For the first resource allocation ($\mu\mathrm{P}_2$), we find the elementary cluster activations γ_I, γ_G1, and $\gamma_\mathrm{D1}\gamma_\mathrm{U1}$. The estimated flexibility as defined by Def. 4 calculates to $f_\mathrm{impl} = 3$. Since our already implemented flexibility is 0 (there is no feasible implementation yet), we try to find feasible implementations for the given elementary cluster activations. With Figure 5 and Table 1, we are able to find a feasible allocation and binding for all elementary cluster activations.

Next, we have to check all timing constraints. Therefore, we define a maximal processor utilization of 69%. If the estimated utilization exceeds this upper bound, we reject the implementation as infeasible. Since the Internet browser need not to meet any timing constraints, the respective implementation is indeed feasible.

For the validation of the digital TV application, we need some more information. As we know the timing constraint imposed on the uncompression and the decryption process, we only need information about how often the authentification and controller processes are executed. The execution of the authentification is scheduled once at system start up. Statistically, the controller process makes up about 0.01% of all process calls in the digital TV application. So, we neglect the authentification and controller process in our estimation. For fulfillment of the performance constraint, the sum of the core execution times of process $\mathrm{P}_{\mathrm{D}}^{1}$ and $\mathrm{P}_{\mathrm{U}}^{1}$ ($95ns + 45ns$) must be less than $0.69 \cdot 300ns$. Evidently, this constraint is met.

Unfortunately, we have to reject the implementation of the game's console application violating the upper utilization bound ($95ns + 90ns \not\leq 0.69 \cdot 240ns$). So our implemented flexibility calculates to $f_{\mathrm{impl}} = 2$ which is still greater than the already implemented flexibility.

Now, we continue with the next possible resource allocation, i.e., $\mu\mathrm{P}_1$. Due to space limitations, we only present the results. The set of Pareto-optimal solutions for this example is given by:

Resources	Clusters	c	f
$\mu\mathrm{P}_2$	$\gamma_{\mathrm{I}}, \gamma_{\mathrm{D1}}, \gamma_{\mathrm{U1}}$	\$100	2
$\mu\mathrm{P}_1$	$\gamma_{\mathrm{I}}, \gamma_{\mathrm{G1}}, \gamma_{\mathrm{D1}}, \gamma_{\mathrm{U1}}$	\$120	3
$\mu\mathrm{P}_2$, G_1, U_2, C_1	$\gamma_{\mathrm{I}}, \gamma_{\mathrm{G1}}, \gamma_{\mathrm{D1}}, \gamma_{\mathrm{U1}}, \gamma_{\mathrm{U2}}$	\$230	4
$\mu\mathrm{P}_2$, D_3, G_1, U_2, C_1	$\gamma_{\mathrm{I}}, \gamma_{\mathrm{G1}}, \gamma_{\mathrm{D1}}, \gamma_{\mathrm{D3}}, \gamma_{\mathrm{U1}}, \gamma_{\mathrm{U2}}$	\$290	5
$\mu\mathrm{P}_2$, A_1, C_2	$\gamma_{\mathrm{I}}, \gamma_{\mathrm{G1}}, \gamma_{\mathrm{G2}}, \gamma_{\mathrm{G3}}, \gamma_{\mathrm{D1}}, \gamma_{\mathrm{D2}}, \gamma_{\mathrm{U1}}, \gamma_{\mathrm{U2}}$	\$360	7
$\mu\mathrm{P}_2$, A_1, D_3, C_1, C_2	$\gamma_{\mathrm{I}}, \gamma_{\mathrm{G1}}, \gamma_{\mathrm{G2}}, \gamma_{\mathrm{G3}}, \gamma_{\mathrm{D1}}, \gamma_{\mathrm{D2}}, \gamma_{\mathrm{D3}}, \gamma_{\mathrm{U1}}, \gamma_{\mathrm{U2}}$	\$430	8

At the beginning, our search space consisted of 2^{25} design points. By calculating the set of possible resource allocations, this design space was reduced to 2^{14} design points. This is, by traversing our specification graph and setting up one boolean equation we are able to reject about 99.9% of our design points as non-Pareto-optimal. After investigating approx. 7000 design points, we have found all 6 Pareto-optimal solutions. For these design points, we estimated the implementable flexibility by solving a single boolean equation. In only approx. 1050 cases (0.0032% of the original search space) the estimated flexibility was greater than the already implemented flexibility. Only for these points, we needed to try to construct an implementation. Hence, our exploration algorithm typically prunes the search space so much that industrial size applications can be efficiently explored within minutes.

Conclusions and Future Work

Based on the concept of hierarchical graphs, we have introduced a formal definition of system flexibility. Furthermore, an algorithm for exploring the flexibility/cost design space was presented. Due to the underlying branch-and-bound strategy, it is possible to prune large regions of a typical search space, while still finding all Pareto-optimal implementations.

In our future work, scheduling will be the main issue of concern. First approaches can be found in [10] and [1]. Pop et al. construct a non-preemptive static scheduling for non-hierarchical process graphs based on deadlines and periods under resource constraints. In [1], first results for exactly scheduling hierarchical dataflow graphs on single processor architectures are presented.

References

[1] B. Bhattacharya and S. Bhattacharyya. Quasi-static Scheduling of Reconfigurable Dataflow Graphs for DSP Systems. In *Proc. of the International Conference on Rapid System Prototyping*, pages 84–89, Paris, France, June 2000.

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

[3] Cadence. *Virtual Component Co-design (VCC)*, 2001. http://www.cadence.com.

[4] K. S. Chatha and R. Vemuri. MAGELLAN: Multiway Hardware-Software Partitioning and Scheduling for Latency Minimization of Hierarchical Control-Dataflow Task Graphs. In *Proc. CODES'01, Ninth International Symposium on Hardware/Software Codesign*, Copenhagen, Denmark, Apr. 2001.

[5] G. D. Hachtel and F. Somenzi. *Logic Synthesis and Verification Algorithms*. Kluwer Academic Publishers, Norwell, Massachusetts 02061 USA, 2 edition, 1998.

[6] C. Haubelt, J. Teich, K. Richter, , and R. Ernst. Flexibility/Cost-Tradeoffs in Platform-Based Design. In *SAMOS – Systems, Architectures, Modeling, and Simulation*, 2002. Will be published in Lecture Notes in Computer Science (LNCS), Vol. 2268, Springer.

[7] C. L. Liu and J. W. Layland. Scheduling Algorithm for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[8] G. D. Micheli. *Synthesis and Opimization of Digital Circuits*. McGraw-Hill, Inc., New York, 1994.

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

[10] P. Pop, P. Eles, T. Pop, and Z. Peng. An Approach to Incremental Design of Distributed Embedded Systems. In *Proc. 38th IEEE/ACM Design Automation Conference (DAC)*, Las Vegas, U.S.A., June 2001.

[11] K. Richter, D. Ziegenbein, R. Ernst, L. Thiele, and J. Teich. Representation of Function Variants for Embedded System Optimization and Synthesis. In *Proc. 36th Design Automation Conference (DAC'99)*, New Orleans, June 1999.

[12] J. Teich. Pareto-Front Exploration with Uncertain Objectives. Proc. First International Conference on Evolutionary Multi-Criterion Optimization, Zurich, Switzerland, Mar. 2001. In Lecture Notes in Computer Science (LNCS), Vol. 1993, pp. 314-328, Springer, 2001.