Improving EA-based Design Space Exploration by Utilizing Symbolic Feasibility Tests

Thomas Schlichter, Christian Haubelt and Jürgen Teich
Department of Computer Science 12
University of Erlangen-Nuremberg, Germany
{schlichter, haubelt, teich}@cs.fau.de

ABSTRACT

This paper will propose a novel approach in combining Evolutionary Algorithms with symbolic techniques in order to improve the convergence of the algorithm in the presence of large search spaces containing only few feasible solutions. Such problems can be encountered in many real-world applications. Here, we will use the example of *design space exploration* of embedded systems to illustrate the benefits of our approach. The main idea is to integrate symbolic techniques into the Evolutionary Algorithm to guide the search towards the *feasible region*. We will present experimental results showing the advantages of our novel approach.

Categories and Subject Descriptors

I.1.2 [**Algorithms**]: Nonalgebraic algorithms; J.6 [**Computer-aided engineering**]: Computer-aided design (CAD)

General Terms

Algorithms, Performance

Keywords

Application, Multi-objective optimization, Speedup technique

1. INTRODUCTION

Today's life is essentially affected by *embedded systems*. Since these systems are embedded in nearly everything around us, we do not recognize embedded systems at first glance. Most popular examples of embedded systems include automotive and telecommunication electronics as well as industrial and domestic automation systems. The goal in embedded system design is to develop an electronic system which meets several constraints imposed on the realization, the so-called *implementation*. Moreover, the implementation is required to be optimized for many objectives simultaneously. Typical objectives in embedded system design are: the costs of a system, its power consumption, its weight, the data throughput, etc.

In order to allow an unbiased search, the task of *design space exploration* is performed before selecting (*decision making*) the actual implementation. Design space exploration is a very challenging constrained multi-objective optimization task [10, 7]. The basic optimization problem is a selection and assignment problem where appropriate hardware resources must be selected for the implementation of the embedded system and the processes which represent the application must be assigned to the selected resources. The task of assigning processes to resources, is known to be $\mathcal{NP}$-complete. Obviously, there is an inherent tradeoff between the number of resources used and the performance of the systems. Many successful design space exploration methodologies based on Evolutionary Algorithms are reported in literature [10, 7, 5, 3, 9]. Due to data dependencies among the processes, nearly all possible implementations are *infeasible*, as will be shown in this paper.

Other *constraint optimization problems* known from literature are often constrained in the objective space only (see e.g. [4, 1]). Our problem is somehow different, as the feasibility of solution depends on the *structure* of the solution [3]. Only if the solution found is feasible, we are able to determine the corresponding *objective values*. Thus, the task of design space exploration is twofold: (i) *guide* the search towards the *feasible region* and (ii) *optimize* the feasible solutions. In this paper, we will focus on the first task. We will present a sophisticated feasibility test using symbolic techniques in order to guide the search. The feasibility test uses Binary Decision Diagrams (BDD) and a technique known as *functional simulation by BDDs* [13]. We will provide experimental results showing the efficiency of our novel method. Note, the proposed idea is not limited to the task of design space exploration of embedded systems.

The rest of the paper is organized as follows: Section 2 contains the definition of the search space, which is represented by a so called *specification graph*. This section also defines the task of design space exploration and the actual optimization problem. In Section 3, we describe how the optimization problem can be solved using Evolutionary Algorithms. Here, the test for feasible solutions as well as our proposed improvements will be discussed in detail. We will provide experimental results showing the advantages of our novel approach in Section 4. Finally, Section 5 will conclude the paper and will show some future research directions.

2. PROBLEM STATEMENT

In this paper, we consider the problem of design space exploration for embedded systems. In contrast to general purpose computers, embedded systems are designed for a particular application while satisfying several constraints. Basically, the design space exploration problem is a multi-objective selection and assignment problem. Moreover, typical problems known from real-world ap-
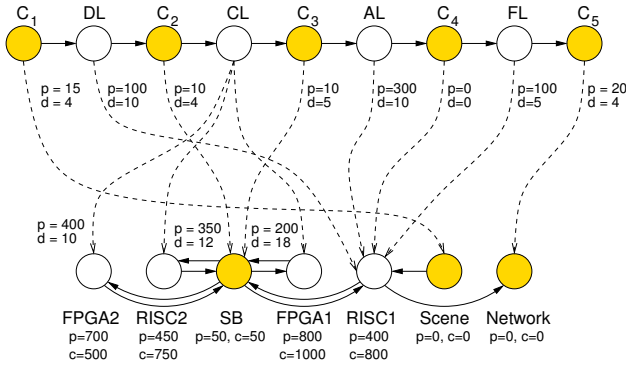
Figure 1: Specification graph for an MPEG4 encoder consisting of process graph g_p and an architecture graph g_a as well as additional mapping edges E_m represented by dashed lines.

plications show search spaces containing only a very small fraction of feasible solutions.

2.1 Defining the Search Space

To allow a mathematical model of the search space, the concept of a so-called *specification graph* is needed. A specification graph specifies an embedded system by means of its applications, architectures, and the relation between these two views. Here, we use a graph-based approach already proposed by Blickle et al. [3]. The specification graph consists of two main components:

- A given application 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 $g_s = (V_s, E_s)$. 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 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 *process graph* $g_p = (V_p, E_p)$ in the following. 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 1 *The example in the upper part of Figure 1 shows an MPEG4 encoder. 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 (Decomposition Layer DL). Each audio/visual object is coded by an appropriate coding algorithm (indicated by the coding layer CL). In the next step (Access Unit Layer AL), the data are provided with time stamps, data type (audio, video), clock references, etc. The FlexMux Layer (Flexible*

Multiplexer FL) allows to group streams with the same QoS (quality of service) requirements. Between two data flow dependent operations, we insert an additional vertex in order to model the required communication ($C_{1...5}$).

The architecture including functional resources and buses can also be modeled by a dependence graph termed *architecture graph* $g_a = (V_a, E_a)$. V_a may consist of two subsets containing functional resources (hardware units like an 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 $e \in E_a$ models a directed link between two resources. All the resources are viewed as *potentially allocatable* components.

Example 2 *The process graph given in the previous example is mapped onto a target architecture shown in the lower part of Figure 1. 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.*

Next, it is shown how user-defined mapping constraints representing possible bindings of processes onto resources can be specified in a graph based model.

Definition 2 (Specification Graph) *A* specification graph *$g_s(V_s, E_s)$ consists of a process graph $g_p(V_p, E_p)$, an architecture graph $g_a(V_a, E_a)$, and a set of mapping edges E_m. In particular, $V_s = V_p \cup V_a$, $E_s = E_p \cup E_a \cup E_m$, where $E_m \subseteq V_p \times V_a$.*

Consequently, mapping edges relate the vertices of the process 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 *Figure 1 shows an example of a specification graph. 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. Only the coding layer (CL) can be executed on the alternative resources RISC2, FPGA1, and FPGA2. The mapping edges $e = (v_p, v_a) \in E_m$ are annotated with delay and power consumption values which arise when operation v_p is executed on resource v_a. Furthermore, all resources in Figure 1 are annotated with allocation cost and power consumption values. 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. The goal of design space exploration is to find optimal solutions which satisfy the specification given by the specification graph. Such a solution is called a *feasible implementation* of the embedded systems. Due to the multi-objective nature of this optimization problem, there is in general more than a single optimal solution.

2.2 System Synthesis

Before discussing the design space exploration in detail, we formalize the notion of a *feasible implementation* (cf. [3]). An implementation, being the result of a system synthesis, consists of three parts:

1. the *allocation* that indicates which elements of the architecture graph are used in the implementation,

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

3. the *schedule* assigning a start time to each operation in the process graph.

Before defining the term *implementation* formally, Blickle et al. [3] introduce the so-called *activation* of vertices and edges:

Definition 3 (Activation) *The* activation *of a specification graph $g_s = (V_s, E_s)$ is a function $a : V_s \times E_s \mapsto \{0,1\}$ that assigns to each edge $e \in E_s$ and to each vertex $v \in V_s$ the value 1 (activated) or 0 (not activated).*

The task of system synthesis is to determine an implementation, i.e., assigning activity values to vertices and edges of the specification graph. An implementation consists of an *allocation*, a *binding*, and a *schedule*. For the sake of simplicity, it is assumed that all vertices $v \in V_p$ and all edges $e \in E_p$ of the process graph g_p are activated subsequently.

Definition 4 (Allocation) *An* allocation α *of a given specification graph g_s is the subset of all activated vertices and edges of the architecture graph g_a, i.e.,*

$$\begin{aligned} \alpha &= \alpha_v \cup \alpha_e, \text{where} \\ \alpha_v &= \{v \in V_a \mid a(v) = 1\} \\ \alpha_e &= \{e \in E_a \mid a(e) = 1\} \end{aligned}$$

Here, α_v denotes the set of allocated vertices and α_e denotes the set of allocated edges.

Definition 5 (Binding) *A* binding β *of a given specification graph g_s is the subset of activated mapping edges E_m, i.e.,*

$$\beta = \{e \in E_m \mid a(e) = 1\}$$

In order to restrict the search space, it is useful to determine the set of *feasible allocations* and *feasible bindings*. A feasible binding guarantees that communications demanded by the process graph can be established in the allocated architecture. This property makes the resulting optimization problem hard to be solved.

Definition 6 (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 activated mapping edge $e \in \beta$ ends at an activated vertex, i.e., $\forall e = (v_p, v_a) \in \beta : v_a \in \alpha$.*

2. *For each process graph vertex $v \in V_p$, exactly one outgoing mapping edge $e \in E_m$ is activated, i.e.,*

$$\left| \{e \in \beta \mid e = (v_p, v_a), v_a \in V_a\} \right| = 1.$$

3. *For each process graph edge $e \in (v_i, v_j) \in E_p$:*

   - *either both operations are mapped onto the same vertex, i.e., $\widetilde{v}_i = \widetilde{v}_j$ with $(v_i, \widetilde{v}_i), (v_j, \widetilde{v}_j) \in \beta$,*
   - *or there exists an activated edge $\widetilde{e} = (\widetilde{v}_i, \widetilde{v}_j) \in E_a \cap \alpha$ in the architecture graph to handle the communication associated with edge e, i.e.,*

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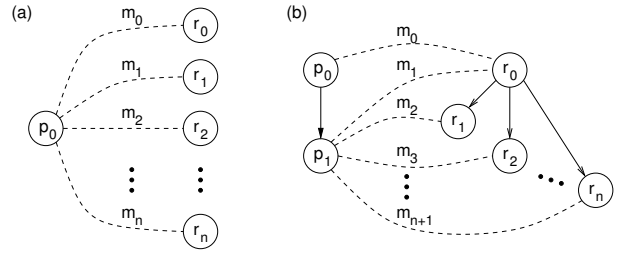

Figure 2: (a) For each process $p \in V_p$ exactly one outgoing mapping edge has to be activated. (b) In order to establish the required communication $(p_0, p_1) \in E_p$, we have to execute the processes p_0 and p_1 on the same resource r_0 or on adjacent resources.

Example 4 *The second and third requirement are depicted in Figure 2: To achieve a feasible binding, the second requirement makes the process p_0 in Figure 2(a) to be bound to exactly one of the allocated resources $r_0 \ldots r_n$. The third requirement is depicted in Figure 2(b). Here, the process p_0 must be bound to the resource r_0 and thus the dependent process p_1 must be bound either to the same resource r_0 or to one of the adjacent resources $r_1 \ldots r_n$.*

Definition 7 (Feasible Allocation) *A* feasible allocation *is an allocation α that allows at least one feasible binding β.*

Finally, a *schedule* can be computed. Therefore, the delay times $delay(v, \beta)$ for each vertex $v \in V_p$ are needed. In general, this delay depends on the current binding of the implementation. In other words, the execution time is affected by the resource an operation is bound to. These delay times are annotated at the mapping edges in the specification graph as described in the preceding section.

Definition 8 (Schedule) *Given a specification graph g_s containing a process graph g_p, a feasible binding β, and a function delay which determines the execution time $delay(v, \beta) \in \mathbb{Z}_{\geq 0}$ of a process graph vertex $v \in V_p$. A* schedule *is a function $\tau : V_p \mapsto \mathbb{Z}_{\geq 0}$ that satisfies for all edges $e = (v_i, v_j) \in E_p$:*

$$\tau(v_j) \geq \tau(v_i) + delay(v_i, \beta)$$

The schedule τ determines the start time of each operation $v_i \in V_p$ where $\tau(v_i)$ represents the start time of operation v_i.

With the above discussion, we can define an *implementation* by means of a feasible allocation, a feasible binding, and a schedule.

Definition 9 (Implementation) *Given a specification graph g_s, a (feasible)* implementation ψ *is a triple (α, β, τ) where α is a feasible allocation, β is a corresponding feasible binding, and τ is a schedule.*

Example 5 *Consider the case that the operation CL in Figure 1 is mapped onto the resource RISC2 as shown in Figure 3(a). The allocation of vertices is given as:*

$$\alpha_v = \{SB, RISC1, RISC2, Network, Scene\}$$

A feasible binding is given by:

$$\begin{aligned} \beta = \{ &(C_1, Scene), (DL, RISC1), (C_2, SB), (CL, RISC2), \\ &(C_3, SB), (AL, RISC1), (C_4, RISC1), (FL, RISC1), \\ &(C_5, Network)\} \end{aligned}$$

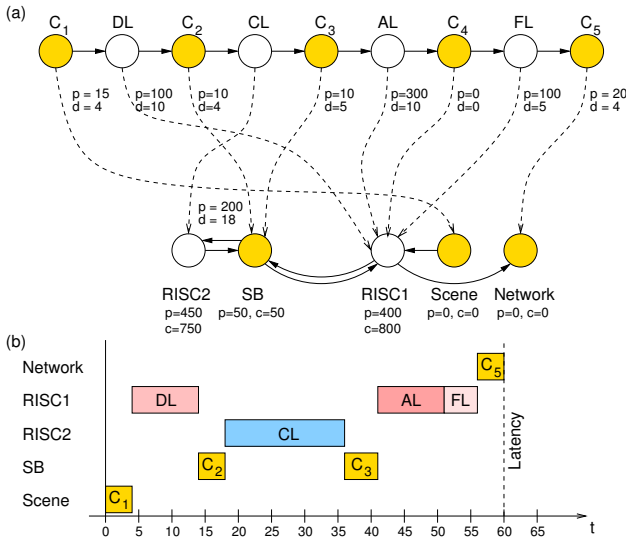Figure 4: The decoding of an individual to an implementation.

Figure 3: (a) Allocation and Binding of the MPEG4 encoder (see Figure 1). (b) Schedule visualized by a Gantt chart.

Given this allocation and binding, one can see that the implementation is indeed feasible, i.e., α and β are feasible. Figure 3(b) shows the schedule τ for this implementation using a so-called Gantt chart.

Given the implementation ψ, some properties of ψ can be calculated. For example, the total cost of the implementation shown in Figure 3 is given by $cost(\psi) = 1600$. The total power consumption power is given by $power(\psi) = 1655$ and the latency of ψ is $latency(\psi) = 60$. The cost and power values are obtained by simply adding the c and p attributes of the activated elements, respectively. The latency results from the computed schedule.

2.3 The Optimization Problem

Now, the task of system synthesis can be formulated as a combinatorial *Multi-objective Optimization Problem* (*MOP*).

Definition 10 (System Synthesis) *The task of* system synthesis *is the following multi-objective optimization problem (see e.g., [15]) where without loss of generality, only minimization problems are assumed here:*

    minimize $f(x)$,
    subject to:
        x represents a feasible implementation $\psi = \psi(x)$,
        $c_i(x) \leq 0, \forall i \in \{1, \ldots, q\}$

where $x = (x_1, x_2, \ldots, x_m) \in X$ is the decision vector, X *is the* decision space, $f(x) = (f_1(x), f_2(x), \ldots, f_n(x)) \in Y$ *is the* objective function *and Y is the* objective space.

Here, x is an encoding called *decision vector* representing an implementation ψ. Moreover, there are q constraints $c_i(x)$, $i = 1, \ldots, q$ imposed on x defining the set of feasible implementations. The *objective function f* is n-dimensional, i.e., n objectives are optimized simultaneously. For example, in embedded system design it is required that the monetary cost and the power dissipation of an implementation are minimized simultaneously. Often, objectives in embedded system design are conflicting [6].

Only those *design points* $x \in X$ that represent a feasible implementation ψ and that satisfy all constraints c_i, are in the set of
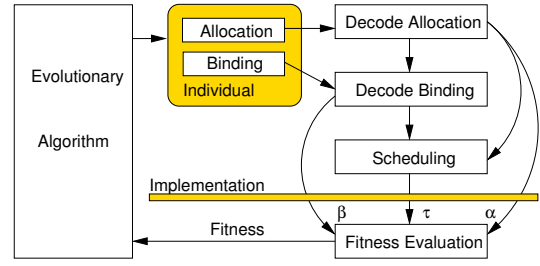
feasible solutions, or for short in the *feasible set* called $X_f = \{x \mid \psi(x) \text{ being feasible } \wedge c(x) \leq 0\} \subseteq X$. The objective space of X is defined as $Y = f(X) \subset \mathbb{R}^n$, where the objective function f on the set X is given by $f(X) = \{f(x) \mid x \in X\}$. Analogously, the *feasible region* in the objective space is denoted by $Y_f = f(X_f) = \{f(x) \mid x \in X_f\}$.

In single-objective optimization, the feasible set X_f is totally ordered, whereas in multi-objective optimization problems, the feasible set X_f is only partially ordered and, thus, there is generally not only one global optimum, but a set of so-called *Pareto points*. A Pareto-optimal implementation is a design point which is not worse than any other feasible solution in the design space in all objectives.

In the following, the necessary definitions for multi-objective optimization problems are given (cf. [15, 11]). Without loss of generality, only minimization problems are considered.

Definition 11 (Pareto dominance) *For any two decision vectors a and b,*

$$a \succ b \ (a \text{ dominates } b) \Leftrightarrow \forall i : f_i(a) \leq f_i(b) \wedge \exists i : f_i(a) < f_i(b)$$

Definition 12 (Pareto optimality) *A decision vector $x \in X_f$ is said to be non-dominated regarding a set $A \subseteq X_f$ iff*

$$\nexists a \in A : a \succ x.$$

A decision vector x is said to be Pareto-optimal iff x is non-dominated regarding X_f.

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 *approximation set X_a* in the following. Furthermore, the *Pareto-optimal front* is given by $Y_p = f(X_p)$.

3. DESIGN SPACE EXPLORATION

In this section, we will show how to solve the system synthesis problem by using Evolutionary Algorithms (EAs). The EA determines the allocation and bindings. Afterwards, the schedule of an implementation is computed by a list scheduler. The main idea is outlined in Figure 4.

3.1 The Evolutionary Algorithm

To obtain a meaningful encoding 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 [3]. 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

```
decoding
   IN:    The individual j consisting of allocation alloc, repair
          allocation priority list $L_R$, binding order list $L_O$, and
          binding priority list $L_B(v)$.
   OUT:   The allocation $\alpha$ and the binding $\beta$ if both are feasible,
          $(\emptyset, \emptyset)$ if no feasible binding is represented by the
          individual j
   BEGIN
      $\bar{\alpha} \leftarrow$ allocation($alloc(j), L_R(j)$)
      $\bar{\beta} \leftarrow$ binding($L_B(j), L_O(j), \bar{\alpha}$)
      IF $\bar{\beta} = \emptyset$
         RETURN $(\emptyset, \emptyset)$
      ENDIF
      $\beta \leftarrow \bar{\beta}$
      $\alpha \leftarrow$ update_allocation($\bar{\alpha}, \bar{\beta}$)
      RETURN $(\alpha, \beta)$
   END
```

Figure 5: Algorithm to decode the allocation α and the binding β from a given individual j.

mechanisms easily. But as the determination of a feasible allocation or binding is $\mathcal{NP}$-complete [3], this would result in solving an $\mathcal{NP}$-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 (cf. Figure 5 for the decoding algorithm):

1. the allocation of resources $v \in V_a$ is decoded from the individual and repaired with a simple heuristic (the function allocation),

2. next the binding of the edges $e \in E_m$ is performed (the function binding), and

3. finally, the allocation is updated in order to eliminate unnecessary vertices $v \in V_a$ from the allocation and all necessary edges $e \in E_a$ are added to the allocation (the function update_allocation()).

Thus, the decoding function results in a feasible allocation and binding of the vertices and edges of the process graph g_p to the vertices and edges of the architecture graph g_a. If no feasible binding could be found, the whole decoding of the individual is aborted.

The allocation of vertices is directly encoded in the so-called *chromosome*, i.e., the elements in a vector *alloc* encode for each vertex $v \in V_a$ if it is activated or not, i.e., $a(v) = alloc(v)$. This simple encoding might result in many infeasible allocations. Hence, a simple repair heuristic is applied. This heuristic only adds new vertices $v \in V_a$ to the allocation and reflects the simplest case of infeasibility that may arise from non-executable functional vertices: Consider the set $V_B \subseteq V_p$ that contains all vertices that can not be executed, because not a single corresponding resource vertex is allocated, i.e., $V_B = \{v \in V_p \mid \forall (v, \widetilde{v}) \in E_m : a(\widetilde{v}) = 0\}$. To make the allocation feasible (in this sense), we add for each $v \in V_B$, at most one $\widetilde{v} \in V_a$, until feasibility in the sense above is achieved.

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

f, the order should automatically be adapted. This will be achieved by the introduction of a *repair allocation priority list L_R* coded in the individual (see, e.g., [14] for a discussion on permutations). In this list, all resources $v \in V_a$ 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.

In this paper, the function binding is of special interest and should be discussed in more detail. A binding is obtained by activating exactly one incident edge $e \in E_m$ for each allocated vertex $v \in V_p$. 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:

All processes are bound in the order they appear in the binding order list L_O. For each process $p \in V_p$, a list L_B is encoded as allele that contains all incident edges $e \in E_m$ to p. 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., $a(e_k) := 1$. The test of feasibility is directly related to the definition of a feasible binding (see Definition 6). As the priority lists contain all incident edges $e \in E_m$, each individual contains a feasible binding iff it contains a feasible allocation. Calculating a feasible binding for a given allocation is equivalent to solve the underlying satisfiability problem. To solve this problem, we can use fast heuristics that try to find a feasible binding, or we use prohibitively slow correct methods. Different methods are described and compared later in this paper. When no feasible binding is found, β is the empty set, and the individual will be given a penalty value as its fitness value. Finally, in the function update_allocation, vertices of the allocation that are not used will be removed from the allocation. Furthermore, all edges $e \in E_a$ in the architecture graph g_a are added to the allocation that are necessary to obtain a feasible binding.

3.2 Feasibility Test

As stated above, finding a feasible binding for a given allocation is a complex task. The binding is performed by binding one process after each other in the order of the binding order list L_O of the individual. When binding a process, the different mapping edges are tested in the order of the binding priority list L_B. To find a feasible binding of the complete problem, only those mapping edges are chosen which satisfy a feasibility test.

This feasibility test can be performed in different ways. The first possibility is to solve the problem whether there still exists a valid completion for all the unbound processes if the currently tested mapping edge is chosen. Of course this will always find the correct binding for each process, and each feasible allocation results in a feasible binding. This problem can be expressed as a boolean satisfiability problem which is satisfiable iff there exists such a completion [8]. The boolean equation for this satisfiability problem can be generated by interpreting the three requirements from Definition 6.

The first requirement states that each activated mapping edge ends at an activated resource. The corresponding boolean equation is:

$$\bigwedge_{p \in V_p : a(p) = 1} \left[\bigvee_{m = (p, r) \in E_m} a(m) \wedge a(r) \right] \quad (1)$$

The second requirement states that exactly one mapping edge is activated for each process. The boolean equation from above already ensures that at least one mapping edge for each process is
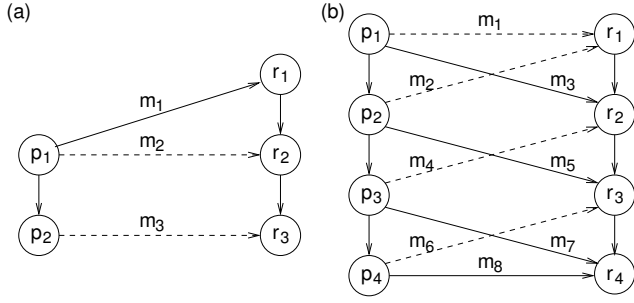
Figure 6: (a) An infeasible binding may result from using the sequential feasibility test whereas using the BDD feasibility test guarantees a feasible solution in this case. (b) Dependencies between four processes which cannot be analyzed even by the BDD feasibility test.

activated. The following equation additionally ensures that at most one mapping edge is activated for each process.

$$\bigwedge_{\substack{m_i=(p,r_i),m_j=(p,r_j)\in E_m:\\ a(p)=1\wedge r_i\neq r_j}} \overline{a(m_i)}\vee\overline{a(m_j)} \qquad (2)$$

The third and last requirement states that communicating processes have to be mapped to the same or to an adjacent resource. This can be expressed by the following equation.

$$\bigwedge_{\substack{m_i=(p_i,r_i)\in E_m:\\ a(p_i)=1\wedge(p_i,p_j)\in E_p}} \left[\overline{a(m_i)}\bigvee_{\substack{m_j=(p_j,r_j)\in E_m:\\ a(p_j)=1\wedge(r_i=r_j\vee(r_i,r_j)\in E_r)}} a(m_j)\right] \qquad (3)$$

As the Equations (1) - (3) have to be satisfied to allow a feasible binding, the complete boolean equation is the conjunction of these. The BDD for this equation can be built once and be used for each individual. When this BDD is built, the test if a mapping edge *m* still allows a feasible binding is just as simple as combining a(*m*) with the BDD with a logic AND-function. So the feasibility test can be done in $O(|E_m|+|V_a|)$ once the BDD is built [13]. Unfortunately, for real-world problems, the BDD is prohibitively large and thus this approach is not viable.

A second possibility is to check if the tested mapping edge is not feasible with the binding performed up to this test. Thus, the mapping edges are activated sequentially and the outcome of this *sequential feasibility test* strongly depends on both, the binding order list L_O and the binding priority list L_B. This test is easy and fast to perform. But as only the yet bound processes are considered, the feasibility test might be trapped, even if the allocation is feasible. This case is sketched in Figure 6(a). If process p_1 is bound to the resource r_1 before binding process p_2, the feasibility test will be passed even though the only feasible binding is $\beta=\{m_2,m_3\}$.

To overcome this drawback, relaxed version of the first feasibility test is proposed here. It is tested if the selection of a certain mapping edge for a process *p* prohibits a feasible binding of any direct predecessor or successor of *p*. It has already been mentioned that encoding the complete boolean equation in one BDD is not feasible for real-world problems. As compromise, we constructed a small BDD for each process. Equations (1) and (2) can be splitted into parts that contain only mapping edges incident to a single process. These boolean equations are encoded in the BDD of the corresponding process. The Equation (3) always contains mapping edges of different processes. Hence, the implementation given by

Equation (3) must be considered in the BDDs associated with p_i and p_j.

With these BDDs, we can test if the activation of mapping edge m_i prohibits the feasibility of the adjacent processes. Therefore, we have to set a(m_i) = 1 for the BDD associated with process p_i belonging to m_i, and for all the BDDs associated with succeeding processes p_j. If one of the BDDs collapses to a logic 0 value, the mapping edge m_i would not allow to find a feasible binding and thus will be rejected.

This method solves the problem shown in Figure 6(a). Of course, this test generally is not able to find a feasible binding for each feasible allocation. This is illustrated in Figure 6(b) where only these bindings are feasible: $\{m_1,m_2,m_4,m_6\}$ and $\{m_3,m_5,m_7,m_8\}$. But as p_1 and p_4 do not have common neighbors, even the BDD feasibility test may select the mapping edges m_1 and m_8 what prohibits a feasible binding.

In this paper, we compare the sequential and BDD feasibility test. Our results show that the BDD feasibility test can improve the convergence even though the test itself is slower. These results are not only interesting for the task of design space exploration, but might also be applied to other combinatorial optimization problems with search spaces containing only few feasible solutions.

4. EXPERIMENTAL RESULTS

In this section, we present experimental results from using the new BDD feasibility test. To analyze the performance of our proposed strategy, we have chosen a MOP from the area of embedded system synthesis. The three objectives used during the optimization are technical properties of embedded systems, namely *implementation cost*, *power dissipation*, and *latency*. In the following, we provide quantitative results from the comparison between the sequential feasibility test and the BDD feasibility test. The PISA (Platform independent Interface for Search Algorithms) [2] framework was chosen for optimization purposes. In the present work, the SPEA2 selection procedure [16] was applied.

The experiments are performed as follows: A generator program is used to randomly construct MOP instances (specification graphs), where several parameters determine the architecture, the problem graph, mapping edges, and the attributes used to compute the objective values. Due to different random values the generated problem instances are similar in structure, but not equal. Each MOP instance is optimized by both methods (with and without BDD feasibility test). After the optimization of each problem instance, the non-dominated solutions X_a found by both methods are combined in a single *reference set* X_R. This reference set is Pareto-filtered and is used to quantitatively assess the performance of both methods.

4.1 Problem Instances and Parameters

The MOP instances (specification graphs) can be generated from a few parameters. The most important ones are: (i) The number of resources *r* in the architecture graph. From these resources a subset must be chosen during optimization. Hence, this number affects the problem size. (ii) The number of processes *p* in the process graph g_p. This number also affects the MOP size. (iii) The number of mapping edges *m* per process. The number of mapping edges has a large influence of the MOP size and complexity as discussed before. (iv) The number of edges in the process graph is given by a probability value d_p. This value determines the probability that two processes are connected by an edge. Due to the feasibility requirement, the number of data dependencies affects the complexity of the optimization problem. The complexity increases with the number of data dependencies. (v) The feasibility probability f_p. This value is used when the edges E_a of the architecture graph are cre-
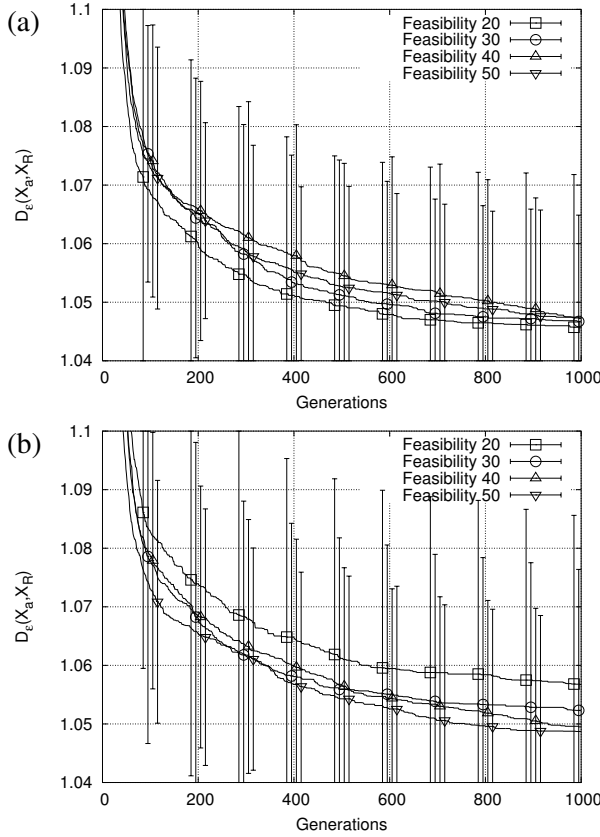
Figure 7: The mean ε-dominance and its standard deviation over the number of generations using (a) the BDD feasibility test and (b) the sequential feasibility test.
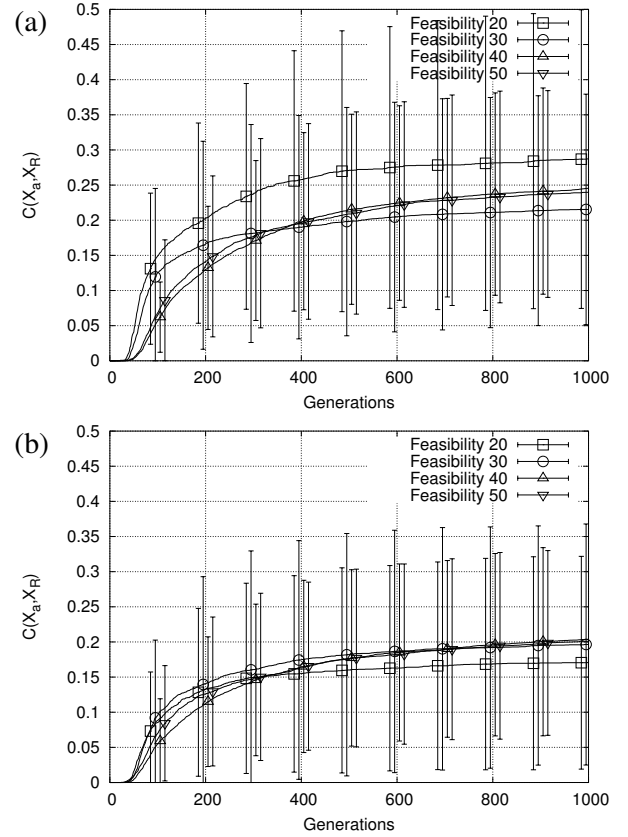

Figure 8: The mean coverage and its standard deviation over the number of generations using (a) the BDD feasibility test and (b) the sequential feasibility test.

ated. For each mapping possibility $m_i = (p_i, r_i), m_j = (p_j, r_j) \in E_m$ of two adjacent processes p_i and p_j, the resources r_i and r_j are connected with probability f_p to satisfy the data dependency. The more of these edges are created, the more feasible solutions exist. Smaller values of f_p result in search spaces containing less feasible solutions.

We created four classes of problem instances with feasibility probabilities $f_p = 20\%, 30\%, 40\%, 50\%$. For each of these problem classes we created 10 different problem instances. The genetic algorithm was run 10 times for each problem instance with both feasibility tests. The different values of f_p make the problem classes to vary in their number of solutions. Problems with a small feasibility value have less solutions than problems with a large feasibility value. The different problem instances all have $p = 45$ processes and $r = 15$ resources. Each process has $m = 2 \ldots 4$ random mapping edges, and $d_p = 30\%$.

The parameters for the EA are chosen as follows: The population size was set to 150. For recombination, 50 children were created from 50 parents by single-point crossover. The mutation rate was set to $|decision\ variables|^{-1}$. The mutation operation is either single bit flip or order-based mutation.

4.2 Qualitative Results

The performance indicators used in the present work are: (i) The *coverage* [17] which measures the fraction of non-dominated points in the reference set X_R found by a particular optimization run (X_a). (ii) The ε-*dominance* [12] where ε is the smallest value such that

the set $X_{a,\varepsilon}$ dominates the reference set X_R. $X_{a,\varepsilon}$ is obtained by scaling each element in the approximation set X_a by the factor $\frac{1}{\varepsilon}$. A detailed discussion on performance indicators can be found in [18]. The approximation sets X_a obtained from both optimization methods are compared to the reference set X_R by using these performance indicators. Moreover, the average time needed for a fix number of generations is calculated as well. The estimation of the consumed processor time is realized by the "clock"-function of the Linux operating system.

Figure 7 shows the mean ε-dominance and its standard deviation for (a) the BDD feasibility test and (b) the sequential feasibility test. The graph shows the average of the 100 runs for each problem class, and the vertical bars indicate the standard deviation. As one can see, for hard problems (small f_p values) the BDD feasibility test makes the EA to converge faster against the optimal value of 1.0 than the sequential feasibility test. Also the standard deviation is smaller with the BDD feasibility test.

Figure 8 shows the mean coverage and its deviation for (a) the BDD feasibility test and (b) the sequential feasibility test. Here, the BDD feasibility test always outperforms the sequential feasibility test. Moreover, the harder the problem is, the better is the BDD feasibility test. In the case of the coverage metric, the standard deviations are comparable.

Figure 7 and Figure 8 show how the different feasibility tests converge over the number of generations. To show the speed difference, the coverage is drawn over the average computation time in Figure 9. As the sequential feasibility test is faster, its calcula-
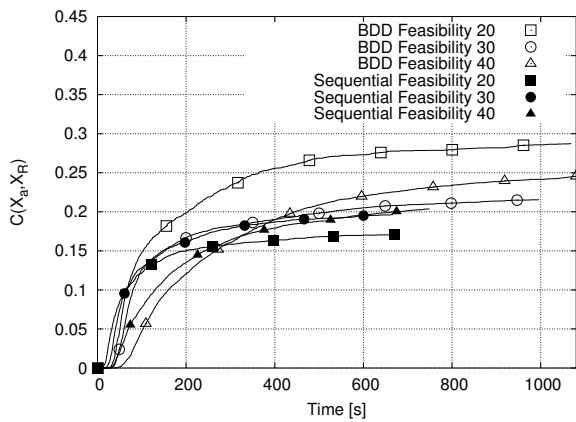
Figure 9: The mean coverage over the time. As the sequential feasibility test is faster than the BDD feasibility test, the calculation of the 1000 generations finishes earlier.

tion of 1000 generations finished earlier. Nevertheless, one can see that the BDD feasibility test is worse in the beginning. This is also due to the time needed for constructing the BDDs. But, in the presence of search spaces containing few feasible solutions only, the BDD feasibility test clearly outperforms the sequential feasibility test after a small amount of time.

5. CONCLUSIONS AND FUTURE WORK

We have introduced a method to improve the convergence of EA-based algorithms for problems with a search space containing many infeasible solutions. This novel approach makes use of symbolic techniques to guide the search of the EA towards the feasible region. We have presented experimental results which clearly show the benefits of our approach. Although we used the example of design space exploration of embedded systems, there is however the possibility to generalize our approach to other constrained combinatorial optimization problems.

In the near future, we will generalize our approach of integrating symbolic techniques into EAs. Moreover, we plan to study our methodology on arbitrary industrial problems.

6. REFERENCES

[1] A. H. Aguirre, S. B. Rionda, C. A. C. Coello, G. L. Lizárraga, and E. M. Montes. Handling Constraints using Multiobjective Optimization Concepts. *International Journal for Numerical Methods in Engineering*, 59(15):1989–2017, Apr. 2004.

[2] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA - A Platform and Programming Language Independent Interface for Search Algorithms. In *Proceedings of the Second International Conference on Evolutionary Multi-Criterion Optimization, Lecture Notes in Computer Science (LNCS)*, volume 2632, pages 494–508, Faro, Protugal, Apr. 2003.

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

[4] K. Deb. *Multi-Objective Optimization using Evolutionary Algorithms*. John Wiley & Sons, Ltd., Chichester, New York, Weinheim, Brisbane, Singapore, Toronto, 2001.

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

[6] M. Eisenring, L. Thiele, and E. Zitzler. Conflicting Criteria in Embedded System Design. *IEEE Design & Test of Computers*, 17(2):51–59, June 2000.

[7] C. Haubelt, S. Mostaghim, F. Slomka, J. Teich, and A. Tyagi. Hierarchical Synthesis of Embedded Systems Using Evolutionary Algorithms. In R. Drechsler and N. Drechsler, editors, *Evolutionary Algorithms for Embedded System Design*, Genetic Algorithms and Evolutionary Computation (GENA), pages 63–104. Kluwer Academic Publishers, Boston, Dordrecht, London, 2003.

[8] C. Haubelt, J. Teich, R. Feldmann, and B. Monien. SAT-Based Techniques in System Design. In *Proceedings of Design, Automation and Test in Europe*, pages 1168–1169, Munich, Germany, Mar. 2003.

[9] X. Hu, G. W. Greenwood, and J. G. D'Ambrosio. An Evolutionary Approach to Hardware/Software Partitioning. In *Proceedings of Parallel Problem Solving from Nature*, pages 900–909, Berlin, Germany, Sept. 1996.

[10] V. Kianzad and S. S. Bhattacharyya. CHARMED: A Multi-Objective Co-Synthesis Framework for Multi-Mode Embedded Systems. In *Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP'04)*, pages 28–40, Galveston, U.S.A., Sept. 2004.

[11] M. Laumanns. *Analysis and Applications of Evolutionary Multiobjective Optimization Algorithms*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Aug. 2003.

[12] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler. Combining Convergence and Diversity in Evolutionary Multi-Objective Optimization. *Evolutionary Computation*, 10(3):263–282, 2002.

[13] C. Scholl, R. Drechsler, and B. Becker. Functional Simulation Using Binary Decision Diagrams. In *Proceedings of the 1997 IEEE/ACM Int. Conference on Computer-Aided Design*, pages 8–12, San Jose, USA, Nov. 1997.

[14] D. Whitley. Permutations. In *Evolutionary Computation 1 – Basic Algorithms and Operators*, pages 139–150. Institute of Physics Publishing, Bristol and Philadelphia, 2000.

[15] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, Eidgenössische Technische Hochschule Zürich, Nov. 1999.

[16] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In *Evolutionary Methods for Design, Optimisation, and Control*, pages 19–26, Barcelona, Spain, 2002.

[17] E. Zitzler and L. Thiele. Multiobjective Optimization Using Evolutionary Algorithms – A Comparative Case Study. In *Proceedings of Parallel Problem Solving from Nature – PPSN-V*, pages 292–301, Amsterdam, The Netherlands, Sept. 1998.

[18] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. Grunert da Fonseca. Performance Assessment of Multiobjective Optimizers: An Analysis and Review. *IEEE Transactions on Evolutionary Computation*, 7(2):117—132, Apr. 2003.