Hybrid Optimization Techniques for System-Level Design Space Exploration

7

Michael Glaß, Jürgen Teich, Martin Lukasiewycz, and Felix Reimann

Abstract

Embedded system design requires to solve synthesis steps that consist of resource allocation, task binding, data routing, and scheduling. These synthesis steps typically occur several times throughout the entire design cycle and necessitate similar concepts even at different levels of abstraction. In order to cope with the large design space, fully automatic Design Space Exploration (DSE) techniques might be applied. In practice, the high complexity of these synthesis steps requires efficient approaches that also perform well in the presence of stringent design constraints. Those constraints may render vast areas in the search space infeasible with only a fraction of feasible implementations that are sparsely distributed. This is a serious problem for metaheuristics that are popular for DSE of electronic hardware/software systems, since they are faced with large areas of infeasible implementations where no gradual improvement is possible. In this chapter, we present an approach that combines metaheuristic optimization with search algorithms to solve the problem of Hardware/Software Codesign (HSCD) including allocation, binding, and scheduling. This hybrid optimization uses powerful search algorithms to determine feasible implementations This avoids an exploration of infeasible areas and, thus, enables a gradual improvement as

M. Glaß
Institute of Embedded Systems/Real-Time Systems at Ulm University, Ulm, Germany
e-mail: michael.glass@uni-ulm.de

J. Teich
Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, Germany
e-mail: juergen.teich@fau.de

M. Lukasiewycz
Robert Bosch GmbH, Corporate Research, Renningen, Germany
e-mail: martin.lukasiewycz@de.bosch.com

F. Reimann
Audi Electronics Venture GmbH, Gaimersheim, Germany
e-mail: felix.reimann@audi.de

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_8

required for efficient metaheuristic optimization. Two methods are presented that can be applied to both, problems with linear as well as non-linear constraints, the latter being particularly intended to address aspects such as timeliness or reliability which cannot be approximated by linear constraints in a sound fashion. The chapter is concluded with several examples for a successful use of the introduced techniques in different application domains.

Acronyms

BIST	Built-In Self-Test
DPLL	Davis-Putnam-Logemann-Loveland
DSE	Design Space Exploration
EA	Evolutionary Algorithm
E/E	Electric and Electronic
ESL	Electronic System Level
HSCD	Hardware/Software Codesign
ILP	Integer Linear Program
MoC	Model of Computation
MPSoC	Multi-Processor System-on-Chip
PB	Pseudo-Boolean
SAT	Boolean Satisfiability
SMT	Satisfiability Modulo Theories

Contents

7.1 Introduction and Motivation

The design of electronic embedded systems typically requires to solve the crucial synthesis steps of resource allocation, task binding, data routing, and scheduling. Those basic steps can even re-occur throughout the design cycle [30] and necessitate similar concepts even at different levels of abstraction. Here, a major problem for

design space exploration is typically not just a vast design space, but the high complexity of these synthesis steps (NP-complete) that becomes even more severe in the presence of stringent design constraints. Those constraints may render many possible system implementations infeasible, such that vast areas in the search space are infeasible with feasible implementations populating the space only sparsely. This is a tremendous problem for metaheuristics that are popular for *Design Space Exploration* (DSE) (see ▸ Chap. 6, "Optimization Strategies in Design Space Exploration"), since they are faced with large areas of infeasible implementations where no gradual improvement is possible.

This chapter describes an approach to overcome this problem. The main idea is to employ a backtracking-based search algorithm, i.e., a *Pseudo-Boolean* (PB) solver, to obtain feasible implementations only. This search algorithm is controlled by a metaheuristic optimization technique, i.e., an *Evolutionary Algorithm* (EA) to (a) enable an efficient exploration even in large and sparse search spaces and (b) search for implementations that are optimized with respect to multiple non-functional design objectives such as timeliness, reliability, and/or power consumption. This combination is termed *SAT decoding* and can be considered a *hybrid optimization* approach which is exemplified for system-level DSE of electronic hardware/software systems in this chapter. However, the employed search algorithm is only capable of handling linear or linearizable design constraints in a Boolean domain.

In the presence of constraints that cannot be efficiently linearized and reduced to the Boolean domain – such as a maximum end-to-end delay of an application with tasks mapped to multiple resources – the approach will again deliver infeasible implementations if these non-linear constraints are ignored. A technique to overcome this drawback is as well presented in this chapter. The basic idea is to integrate analysis techniques for such non-functional constraints and incrementally determine linear constraints for the pseudo-Boolean solver. By this way, the solver is capable of learning which implementations are infeasible. The proposed hybrid Design Space Exploration (DSE) approach is not only applicable at the Electronic System Level (ESL), but may be applied also at other levels of hardware and software synthesis in embedded system design.

This chapter is structured into three main sections: Sect. 7.2 introduces required fundamentals as well as the mathematical formulation of the synthesis problem to be solved. Section 7.3 presents the hybrid optimization technique SAT decoding that can consider linear constraints. An extension of SAT decoding which considers non-linear constraints termed *SMT decoding* is discussed subsequently in Sect. 7.4. Examples of applications of the introduced techniques for further reading are presented in Sect. 7.5 before the chapter is concluded in Sect. 7.6.

7.2 Fundamentals and Problem Formulation

The first work on Hardware/Software Codesign (HSCD) can be found in [22] which considers the problem of concurrently defining a multi-processor system's topology, a binding of tasks to processors, and their scheduling. Since, the problem of allocating hardware and software components, followed by binding tasks to

either hardware or software, became known as *hardware/software codesign*. Many initial works consider the codesign problem a *bipartition problem*, i.e., a task is assigned either to a processor and executed as software or to one dedicated hardware accelerator. This notion is generalized to heterogeneous hardware/software architectures with multiple components in [29] under the term *system-level synthesis*. In [1], same authors prove that this system-level synthesis problem is an NP-complete problem. For an in-depth discussion of the historical roots of hardware/software codesign, see ▸ Chap. 1, "Introduction to Hardware/Software Codesign". For comprehensive overviews on system-level synthesis techniques, interested readers can refer to [6, 28].

In the following sections, we introduce a well-established model for the system-level synthesis problem which is very suitable for (networked) embedded systems. Afterward, we also discuss the problem from an optimization perspective where system synthesis can be considered a *constrained combinatorial optimization problem* and give a brief introduction of common optimization approaches and constraint-handling techniques.

7.2.1 System Model and the System-Level Synthesis Problem

In the following section, we introduce a formal graph-based system model and the respective system-level synthesis problem as a variation and extension of the original model from [1] that is proposed in [21].

7.2.1.1 System Model

A system model ς termed *specification* is given in a graph-based fashion that distinguishes between *application* (modeled as an *application graph* G_T) and an *architecture* (modeled as an *architecture graph* G_R). The relation between application and architecture – indicating each possible binding of a task of the application for execution on a resource of the architecture – is modeled by means of a set of *mapping edges* E_M.

- The application is given by a bipartite directed graph $G_T(T, E_T)$ with $T = P \cup C$. The vertices T are either process tasks $p \in P$ or communication tasks $c \in C$. Each edge $e \in E_T$ connects a vertex in P to one in C, or vice versa. Each process task $p \in P$ can have multiple incoming edges since it might receive data from multiple other process tasks. A process task can also have multiple outgoing edges to model the sending of data to multiple process tasks. The data itself is not directly sent to other processing tasks, but the transmission is modeled explicitly by communication tasks. Each communication task $c \in C$ has exactly one predecessor process task as the sender, since data is sent by exactly one sender. To allow multicast communication, each communication task can have multiple successor process tasks.
- The architecture is modeled as a directed graph $G_R(R, E_R)$. The vertices R represent resources such as processors, memories, or buses. The directed edges $E_R \subseteq R \times R$ indicate available communication connections between resources.

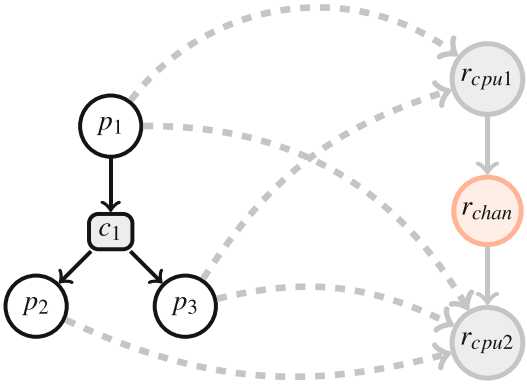


Fig. 7.1 Specification with the application graph G_T on *the left*, the architecture graph G_R on *the right*, and mapping edges depicted *dashed*

- The set of mapping edges E_M contains the mapping information for each process task. Each mapping edge $m = (p, r) \in E_M$ indicates a possible implementation/execution of process $p \in P$ on resource $r \in R$.

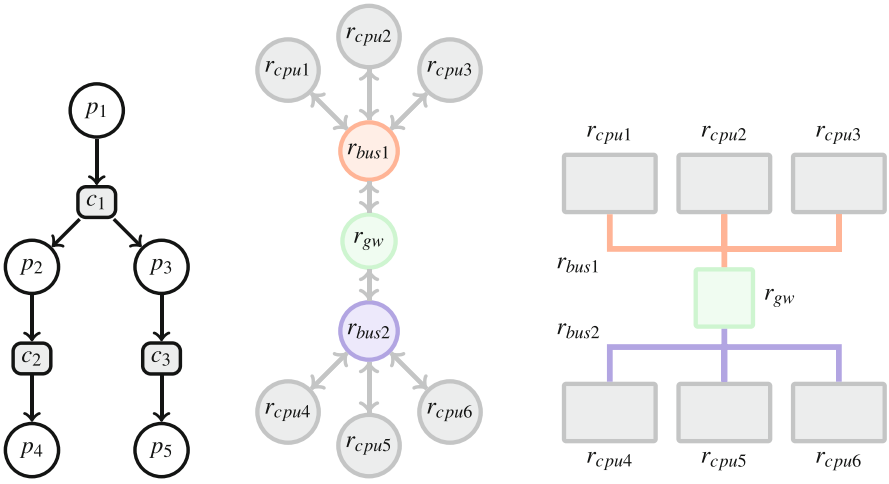
A simple specification including application graph, architecture graph, and mapping edges is given in Fig. 7.1: The application consists of three process tasks (p_1, p_2, and p_3) and one communication task (c_1) that distributes the data produced by p_1 to p_2 and p_3 in a multicast fashion. The architecture consists of two CPU resources capable of executing process tasks (r_{cpu1} and r_{cpu2}) that are connected via a channel (r_{chan}) that only allows a communication from r_{cpu1} to r_{cpu2} but not vice versa as specified by the directed edge $(r_{cpu1}, r_{cpu2}) \in E_R$. Moreover, mapping edges depict which process task can be executed on which resources, i.e., p_1 on r_{cpu1} and r_{cpu2}, p_2 on r_{cpu2}, and p_3 on r_{cpu1} and r_{cpu2}.

A more complex specification is given in Fig. 7.2: The application graph shown in Fig. 7.2a consists of five process tasks and three communication tasks. The architecture graph shown in Fig. 7.2b consists of six processors (CPUs) and two buses that are coupled via a gateway resource. For the sake of brevity and better visualization, we also show an architecture diagram in Fig. 7.2c where processors are depicted as rectangles and buses are depicted as edges between processors and (possibly) gateway components.

7.2.1.2 System-Level Synthesis

The introduced specification is the base for the following formulation of the system-level synthesis problem:

> System-level synthesis derives an *implementation* from a given specification by means of an *allocation* of resources, a *binding* of process tasks to allocated resources, a *routing* of communication tasks on a tree of allocated resources, and a *schedule* of tasks.



(a) $G_T(T, V_T)$ (b) $G_R(R, E_R)$ (c) architecture diagram

Fig. 7.2 A specification with (**a**) application graph G_T and (**b**) architecture graph G_R with mapping edges (not depicted) being $E_M = \{(p_1, r_{cpu1}), (p_1, r_{cpu2}), (p_2, r_{cpu5}), (p_3, r_{cpu2}), (p_3, r_{cpu4}), (p_4, r_{cpu4}), (p_4, r_{cpu6}), (p_5, r_{cpu1}), (p_5, r_{cpu3})\}$. A more compact representation of the modeled architecture as an architecture diagram is given in (**c**)

Formally, an implementation ω consists of the *allocation graph* G_α that is deduced from the architecture graph G_R, the *binding* E_β as a subset of E_M that describes the mapping of the process tasks to allocated resources, the *routing* γ that contains a directed routing graph $G_{\gamma,c}$ for each communication task $c \in C$, and the *schedule function* S. For an implementation to be *feasible*, several conditions have to be fulfilled by the allocation, binding, routing, and scheduling:

- The allocation is a directed graph $G_\alpha(\alpha, E_\alpha)$ that is an induced subgraph of the architecture graph G_R. The allocation contains all resources $r \in R$ that are selected for the current implementation. E_α describes the set of allocated communication connections that are induced from the graph G_R such that $e = (r, \tilde{r}) \in E_\alpha$ if and only if $r, \tilde{r} \in \alpha$.
- The binding $E_\beta \subseteq E_M$ describes the mapping of the process tasks to allocated resources. Here, the following requirements must be fulfilled:
  - Each process task $p \in P$ of the application is bound to exactly one resource:

$$\forall p \in P : |\{m | m = (p, r) \in E_\beta\}| = 1 \tag{7.1}$$

    where $|\cdot|$ denotes the cardinality.
  - Each process task $p \in P$ can only be bound to an allocated resource:

$$\forall m = (p, r) \in E_\beta : r \in \alpha \tag{7.2}$$

- Each communication task $c \in C$ is routed on a tree $G_{\gamma,c} = (R_{\gamma,c}, E_{\gamma,c})$. The routing must be performed such that the following conditions are satisfied:
  - The directed tree $G_{\gamma,c}$ is a connected subgraph of the allocation G_α such that

$$R_{\gamma,c} \subseteq \alpha \text{ and } E_{\gamma,c} \subseteq E_\alpha. \tag{7.3}$$

  - For each communication task $c \in C$, the root of $G_{\gamma,c}$ has to equal the resource on which the predecessor sender process task $p \in P$ is bound:

$$\forall (p,c) \in E_T, m = (p,r) \in E_\beta : |\{e | e = (\tilde{r}, r) \in E_{\gamma,c}\}| = 0 \tag{7.4}$$

    Here, r is the resource to which the sender process task $p \in P$ is bound and by requiring that the number of incoming edges to this node is 0, the node is ensured to be the routing tree's root.
  - For each communication task $c \in C$, $R_{\gamma,c}$ must contain all resources on which any successor process task $p \in P$ is bound:

$$\forall (c,p) \in E_T, m = (p,r) \in E_\beta : r \in R_{\gamma,c} \tag{7.5}$$

- On each resource, different scheduler types may be present to schedule tasks bound to them. Here, a general scheduling approach is considered which assigns each mapping of a (process and communication) task a priority:

$$S : E_M \cup C \rightarrow \{1, \ldots, |T| + |C|\}. \tag{7.6}$$

  In case resources either execute process tasks or route communication tasks, the number of required priorities decreases to $\max(|T|, |C|)$.

  It is then the responsibility of the scheduler to consider the assigned priorities. However, the following two conditions typically have to be fulfilled:
  - Process task priorities are unique per resource $r \in R$:

$$\forall r \in R, m = (t,r), m' = (t',r) \in E_\beta, m \neq m' : S(m) \neq S(m') \tag{7.7}$$

  - Communication task priorities are unique (since they may share several allocated resources on their route):

$$\forall r \in R, c, c' \in C, c \neq c' : S(c) \neq S(c') \tag{7.8}$$

Two implementations for the simple specification given in Fig. 7.1 are shown in Fig. 7.3. The implementation depicted on the left is infeasible since c_1 cannot be routed to p_3. The implementation depicted on the right adheres to all requirements and is, thus, a feasible implementation. Moreover, a feasible implementation for the more complex specification from Fig. 7.2 is given in Fig. 7.4.

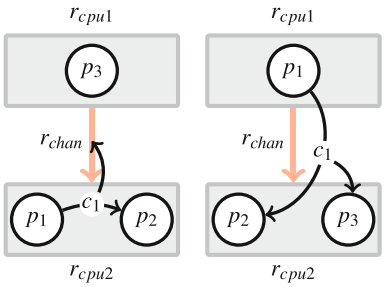


Fig. 7.3 Two implementations of the specification in Fig. 7.2. Shown are two allocated resources r_{cpu1} and r_{cpu2} as *rectangles* and the unidirectional communication channel r_{chan} as a *directed edge* between them. The left implementation is infeasible: Because of the unidirectional communication channel, there exists no connected subgraph to route the communication task c_1 to receiver p_3. The right implementation is feasible

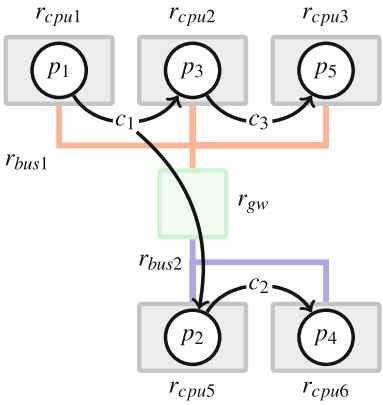


Fig. 7.4 An implementation for the specification in Fig. 7.2. Illustrated is the allocation G_α, the binding E_β of process tasks, and the routing γ of communication tasks. All routings are performed within multiple hops using the available bi-directional buses r_{bus1}, r_{bus2} and the gateway r_{gw}. The communication c_1 is of type multicast

The outlined system-level synthesis problem is typically represented by means of the Y-chart [5] where the separation of application and architecture resulting in an implementation forms a Y as depicted in Fig. 7.5. In general, system-level synthesis shall obviously only deliver feasible implementations. Yet, our aim is to search for implementations that are optimized with respect to multiple and even conflicting design objectives. For this purpose, the Y-chart approach is extended in [1] and later in [12] by a DSE phase that can be seen as an optimization in order to obtain high-quality implementations. This directly brings us to the core topic of this chapter: How to efficiently perform a DSE which has to solve the complex system-level synthesis problem for each considered implementation? Before we come to the introduction of the hybrid optimization technique, we briefly review optimization techniques that are suitable for such kind of problems.

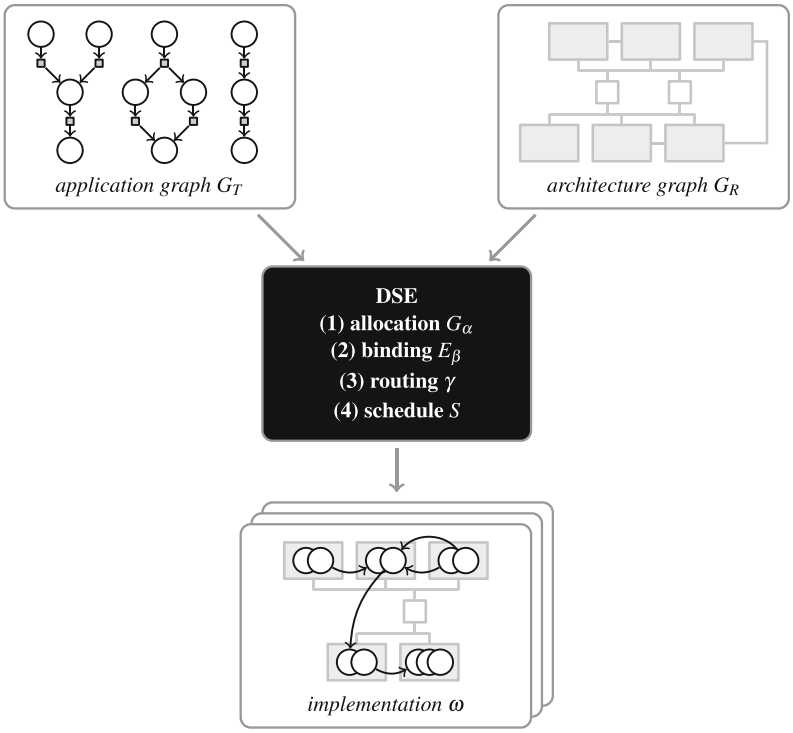


Fig. 7.5 Illustration of the Y-chart approach: An implementation shall be synthesized from a given application graph and architecture graph. For each implementation that is considered during DSE, resource allocation, process task binding, communication task routing, and task scheduling have to be determined with the overall goal to determine a set of Pareto-optimal implementations

7.2.2 Constrained Combinatorial Optimization

As we have seen, whether an implementation that is deduced from a specification is feasible requires it to fulfill the introduced constraints. Also, we can recognize that allocation, binding, routing, and scheduling are all design steps that basically solve combinatorial problems of assigning tasks to resources or priorities to tasks. Thus, we can conclude that system-level synthesis as introduced falls in the class of *constrained combinatorial problems*:

Definition 1 (Constrained Combinatorial Problem).

$$\begin{aligned} &\textit{minimize } f(\mathbf{x}) \\ &\textit{subject to:} \\ &\quad a_i(\mathbf{x}) \leq b_i, \forall i \in \{1, \ldots, q\} \text{ with } b_i \in \mathbb{Z} \end{aligned}$$

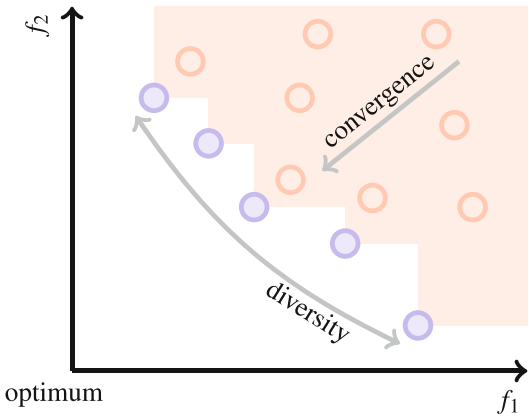


Fig. 7.6 A two-dimensional objective space: Pareto-optimal implementations are depicted *light blue* while dominated implementations are depicted *light red*. The area which is dominated by Pareto-optimal implementations is depicted *light red* as well. Multi-objective optimization approaches try to achieve high convergence as well as high diversity among the found implementations

In system-level synthesis, the objective function f is typically multi-dimensional to consider multiple objective functions such as area, power, or timeliness that can, in particular, be non-linear. Note that in multi-objective optimization problems, there is generally not only one global optimum, but also a set of so-called *Pareto-optimal implementations* is derived with each Pareto-optimal implementation being better in at least one objective when compared to each other feasible implementation. In multi-objective optimization, the notion of one implementation being better than another is given by the concept of *dominance* ($\prec$), i.e., one implementation dominates another if it is better in at least one design objective. Figure 7.6 visualizes this in the objective space with two design objectives f_1 and f_2. Pareto-optimal implementations are depicted light blue and dominated implementations light red. Also, the areas in the objective space that are dominated by a Pareto-optimal implementation are depicted. An indicator for the quality of a multi-objective optimization approach is *convergence*, i.e., how close are the found implementations to the front or Pareto-optimal implementations, and *diversity*, i.e., how well distributed are the implementations in the *objective space*.

For this chapter, focus is not put on the handling of multiple objectives. Instead, the main problem addressed arises from the notion of the *feasible search space* $X_\mathrm{f} \subseteq X$. In the general form, the search space is constrained by q so-called *constraint functions* a imposed on an implementation, formulated as inequalities $a_i(\mathbf{x}) \leq b_i$. The effect of these restrictions is sketched in Fig. 7.7: While every point in the search space of an unconstrained combinatorial problem is feasible, the search space $X_\mathrm{f} \subseteq X$ may contain significantly less or – in the extreme case – even no feasible implementation at all.

In order to solve this problem efficiently, the search space and the types of constraints will be restricted in the following: The search space $X = \{0,1\}^n$ is encoded as a set of two Boolean vectors. Moreover, the constraints are restricted to a single matrix inequation as follows:

$$A\mathbf{x} \leq b \text{ with } \mathbf{x} \in \{0,1\}^n, A \in \mathbb{Z}^{m,n}, b \in \mathbb{Z}^m \tag{7.9}$$

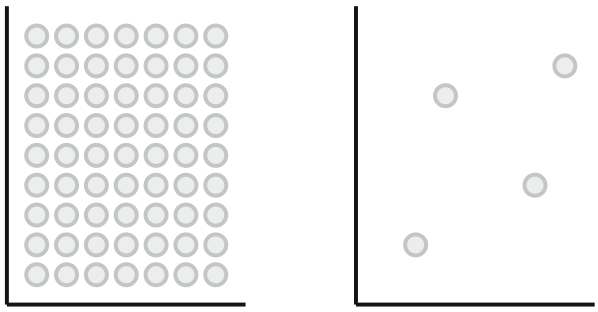


Fig. 7.7 A visualization of the search space of a combinatorial problem (*left*) and a constrained combinatorial problem (*right*) when projecting the search space $X = \{0, 1\}^n$ onto two dimensions: In the unconstrained case, every point in the search space is a feasible implementation while in the constrained case, large areas in the search space might not contain any feasible implementation

As can be seen, the constraints given by $A\mathbf{x} \leq b$ have to be linear or linearizable. In the two main parts of this chapter, we will first discuss that the introduced constraints for allocation, binding, routing, and scheduling can be linearized and, thus, directly be included in the search space. Afterward, we will also introduce how to take care of constraints that cannot be linearized.

As outlined in ▸ Chap. 6, "Optimization Strategies in Design Space Exploration", metaheuristic optimization techniques have become state-of-the-art to solve several problems from the area of hardware/software codesign. This is mainly due to their ability to consider multiple conflicting and even non-linear design objectives. In contrast, exact approaches like Integer Linear Programs (ILPs) require the objective function to be linear and multiple objectives are – in general (see [15]) – not supported. However, applying metaheuristic optimization techniques to constrained combinatorial problems as given by system-level synthesis raises a significant problem: How to determine the set of Pareto-optimal feasible implementations and – in case of really stringent constraints – how to even find one single feasible implementation?

The basic idea behind the relevant metaheuristic approaches is to *vary* selected (high-quality) implementations in an iterative loop and to keep the best found so far in an *archive* $\mathbb{A}$ of non-dominated implementations. In every iteration, varied implementations, e.g., ω, are compared to the ones from the archive, e.g., $\tilde{\omega}$, and – also depending on the concrete metaheuristic optimization technique – either dropped in case they are dominated ($\prec$) by implementations from the archive, i.e., $\omega \prec \tilde{\omega}$, or the archive is updated with the new implementations. This way, metaheuristics gradually but efficiently search for the best implementations. This principle is depicted in Fig. 7.8. However, in the presence of stringent constraints, Fig. 7.9 (left) outlines the effect of variation. The next feasible implementation may be out of reach and since all surrounding implementations are infeasible, only a very slow convergence toward the optimal implementations is achieved. In some cases, it might even occur that not even a single feasible implementation is found.

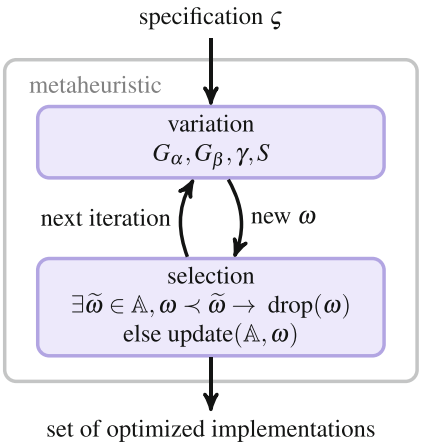


Fig. 7.8 Principle of metaheuristic optimization approaches for hardware/software codesign: Given a specification, the heuristic performs an iterative optimization loop where it varies the allocation, binding, routing, and schedule to explore implementations and selects which implementations (**a**) are to be dropped since they are dominated by implementations from an archive, (**b**) update the archive in case they dominate implementations in the archive, and (**c**) are promising candidates for variation in the next iteration. At the end, a set of optimized (near Pareto-optimal) implementations is the output

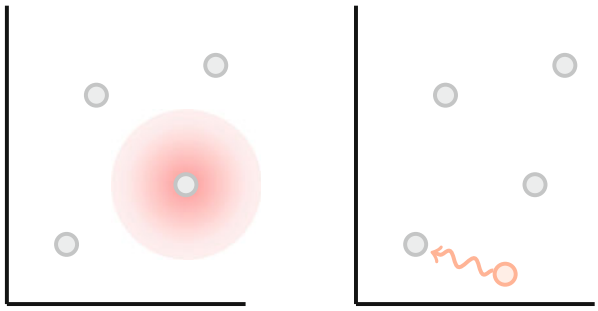


Fig. 7.9 Varying a feasible implementation as a common concept of most metaheuristic optimization techniques may only result in neighboring implementations that are all infeasible (*left*). In the presence of a repair strategy, an infeasible implementation is modified to – if possible – become a feasible implementation (*right*)

As a remedy, *constraint-handling techniques* have been successfully developed to apply metaheuristic optimization techniques to constrained combinatorial problems; see [2] for a comprehensive overview. Here, we will just introduce two main concepts: *Penalty functions* and *repair strategies*. The idea of penalty functions [27] is to transform the constrained problem into an unconstrained problem by deteriorating the original objective function by a penalty function. The amount of penalization depends on the violation of constraints. A similar idea is to leave the original objective function as is and add constraints as additional objectives [11], e.g., minimizing the number of violated constraints. The drawbacks

of these approaches are the increased complexity of the problem by additional objectives and a slow convergence if the feasible region is relatively small compared to the entire search space.

For some combinatorial problems, repair algorithms or at least repair heuristics are available that restore the feasibility of the implementation by applying certain modifications. One well-known problem where such a solution exists is the *0/1 Knapsack Problem* [32] where an infeasible implementation can be repaired by removing items from the knapsack. The idea of the repair strategy is depicted in Fig. 7.9 (right). Since the introduced system-level synthesis problem is NP-complete, also a repair heuristic is, in general, NP-complete and, thus, does not offer a conclusive alternative for our outlined problem.

The rest of this chapter introduces a solution to this problem by means of a hybrid optimization approach.

7.3 Hybrid Optimization

This section introduces a hybrid optimization technique for system-level DSE of embedded systems. First, the key idea of the hybrid optimization technique termed *SAT decoding* – the combination of a metaheuristic with a backtracking-based search algorithm (solver) to consider only feasible implementations during DSE – is presented. Afterward, the main required ingredients to realize such an approach, i.e., (a) the branching strategy of the solver and (b) the formulation of the system-level synthesis problem by means of pseudo-Boolean constraints, are explained.

7.3.1 SAT Decoding: The Key Idea

In the literature, one can find various hybrid optimization approaches that combine heuristic algorithms with exact approaches, also for combinatorial problems from diverse domains; cf. [23] for an overview. The approach discussed in the chapter at hand termed *SAT decoding* [16] falls into the category of *integrative* hybrid optimization approaches. Here, a metaheuristic algorithm is responsible to control the overall optimization procedure: It controls the optimization loop and selects implementations based on their multiple and even non-linear design objectives. In contrast to standard metaheuristics, it does not directly vary the implementation (i.e., allocation, binding, routing, and scheduling). Instead, it integrates a solver – in our case a *Pseudo-Boolean* (PB) solver – that is only responsible to gather feasible implementations, but does not perform any optimization by itself. Following is the key idea of SAT decoding:

> In SAT decoding, instead of varying the implementation directly, the metaheuristic varies the *branching strategy* of the backtracking-based solver. This way, only feasible implementations are obtained and are evaluated during design space exploration.

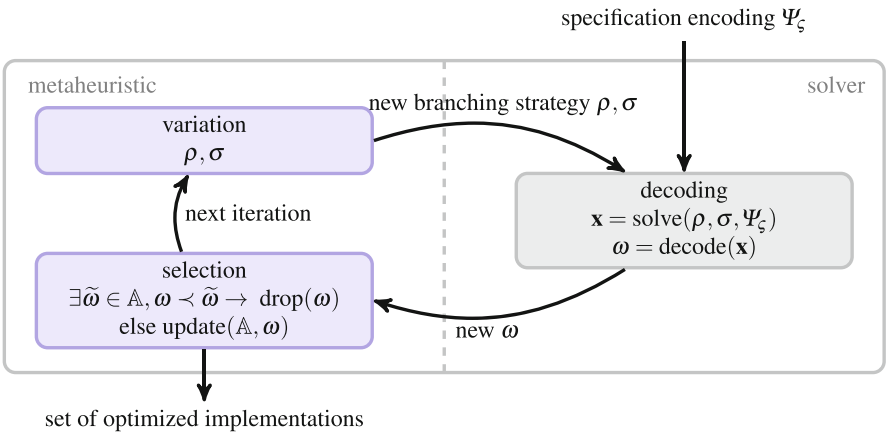


Fig. 7.10 Principle of the SAT-decoding approach: The metaheuristic does not vary the implementation directly but rather varies the parameters σ, ρ of the branching strategy of the employed solver. The solver takes an encoding Ψ_ς of the specification ς together with the current branching strategy and determines a feasible implementation. Each feasible implementation is then delivered to the selection step of the metaheuristic. Note that this hybrid optimization approach only derives feasible implementations to the metaheuristic, circumventing the outlined problems of other state-of-the-art design space exploration approaches for system-level synthesis

Figure 7.10 shows the idea and the resulting integrative hybrid optimization approach. The key feature of this approach is that it combines the advantages of the metaheuristic, i.e., being able to consider multiple and non-linear design objectives, with those of the solver, i.e., only obtaining feasible implementations. After this informal introduction of the approach, each fundamental ingredient is presented in a more detailed fashion in the following subsection.

7.3.2 Solver

To understand how the metaheuristic can actually control the solver to explore the whole diversity of different implementations, we first need to understand how the solver finds feasible implementations – or better – how it solves given search problems in general. Thus, this subsection first introduces the so-called *Pseudo-Boolean* (PB) problem and then shows how most existing solvers approach them by means of the *Davis-Putnam-Logemann-Loveland* (DPLL) backtracking algorithm.

In general, the task of a PB solver is to find a *variable assignment* $\mathbf{x}$ that satisfies a set of linear constraints, i.e., $\mathbf{x} \in X_\text{f}$ as formulated in Equation (7.9). Constraints that are given as linear inequalities and Boolean variables with integer coefficients are known as *PB constraints*. Any ILP solver is also capable of solving PB problems. But, specialized *PB solvers* tend to outrun ILP solvers on Pseudo-Boolean problems because they are based on efficient backtracking algorithms. In fact, many PB solvers are extended Boolean Satisfiability (SAT) solvers with the capability to handle PB constraints and rely on the Davis-Putnam-Logemann-

Loveland (DPLL) algorithm [3]. To comprehend the SAT-decoding technique requires an understanding of the main concept of the DPLL algorithm which is outlined in Algorithm 2. The algorithm starts with a set of completely unassigned

Algorithm 2 DPLL backtracking algorithm

```
1:  procedure SOLVE(σ, ρ)
2:      while true do
3:          branch(σ, ρ)
4:          if CONFLICT then
5:              backtrack()
6:          else if SATISFIED then
7:              return x
8:          end if
9:      end while
10: end procedure
```

variables. Then, the operation branch(σ, ρ) selects an unassigned variable $\mathbf{x}_i$ and assigns it the value 0 or 1 (Line 3). Which variable is selected and which value (0 or 1) is assigned is called *branching strategy*. The branching strategy – which is of key importance for the SAT-decoding approach – is guided by the vectors $\sigma \in \{0,1\}^n$ and $\rho \in \mathbb{R}^n$: Unassigned variables $\mathbf{x}_i$ with the highest value ρ_i are prioritized and set to the value σ_i. Of course, as is common in all backtracking-based solvers, arising conflicts are recognized and resolved (Line 4). A conflict is recognized if any constraint is not satisfiable anymore and *backtracking* is triggered (Line 5). Backtracking means that variable assignments made before are reverted. When all variables have a variable assignment and no conflict occurs (Line 6), then the variable assignment is a feasible solution to the specified problem and returned (Line 7). The majority of the state-of-the-art PB solvers like SAT4J [13] are based on the DPLL algorithm.

Knowing the algorithm of the solver, we can draw two important conclusions: First, we can control which variable assignment, i.e., which feasible implementation, is delivered by the solver by varying the two vectors σ and ρ of the branching strategy. Thus, the *search space* of the metaheuristic in SAT decoding is not the allocation G_α, the binding E_β, the routing γ, and the schedule S, but is solely given by the two vectors σ and ρ of the solver's branching strategy. Second, we have to find an *encoding* Ψ_ς of a specification ς and the introduced system-level synthesis problem by means of pseudo-Boolean constraints such that each feasible variable assignment $\mathbf{x}$ represents a feasible implementation of a given specification.

7.3.3 Pseudo-Boolean Encoding of Allocation, Binding, Routing, and Scheduling

In the following, we present a pseudo-Boolean encoding Ψ_ς of a specification ς and the system-level synthesis problem such that a solution $\mathbf{x} \in \{0,1\}^n$ corresponds to a *feasible* implementation ω according to Equations (7.1), (7.2), (7.3), (7.4), (7.5),

(7.6), (7.7), and (7.8). First, we introduce the required Boolean variables used to formulate the linear constraints:

r
: A Boolean variable for each resource $r \in R$. It indicates whether the resource is allocated $r \in \alpha$ (1) or not (0).

m
: A Boolean variable for each mapping edge $m \in E_M$. It indicates whether the mapping edge is part of the binding, i.e., $m \in E_\beta$ (1) or not (0).

$\mathbf{c_r}$
: A Boolean variable for each communication task $c \in C$ and resource $r \in R$. It indicates whether the communication task c is routed over the resource r (1) or not (0)

$\mathbf{c_{r,\tau}}$
: A Boolean variable for each communication $c \in C$ and resource $r \in R$. It indicates at which communication step $\tau \in \mathcal{T} = \{1, .., |\mathcal{T}|\}$ a communication is routed over the resource. Note that communication tasks are propagated in steps or hops, respectively.

With these variables, we can formulate the linear constraints that encode all introduced requirements in Equations (7.1), (7.2), (7.3), (7.4), (7.5), (7.6), (7.7), and (7.8) for a feasible implementation. First, we start with the linear constraints regarding allocation, binding, and routing:
$\forall p \in P$:

$$\sum_{m=(p,r)\in E_M} \mathbf{m} = 1 \tag{7.10}$$

$\forall m = (p, r) \in E_M$:

$$\mathbf{r} - \mathbf{m} \geq 0 \tag{7.11}$$

The constraints in Equations (7.10) and (7.11) ensure that each task is bound to exactly one resource (cf. Equation (7.1)) and that this resource is allocated (cf. Equation (7.2)). Exemplarily, we show the constraints that would result from Equations (7.10) and (7.11) for the simple specification given in Fig. 7.1:

$$\begin{aligned}
\mathbf{m}_{p_1,r_{cpu1}} + \mathbf{m}_{p_1,r_{cpu2}} &= 1 \\
\mathbf{m}_{p_2,r_{cpu2}} &= 1 \\
\mathbf{m}_{p_3,r_{cpu1}} + \mathbf{m}_{p_3,r_{cpu2}} &= 1 \\
\mathbf{r}_{cpu1} - \mathbf{m}_{p_1,r_{cpu1}} &\geq 0 \\
\mathbf{r}_{cpu1} - \mathbf{m}_{p_3,r_{cpu1}} &\geq 0
\end{aligned}$$

$$\mathbf{r}_{cpu2} - \mathbf{m}_{p_1,r_{cpu2}} \geq 0$$
$$\mathbf{r}_{cpu2} - \mathbf{m}_{p_2,r_{cpu2}} \geq 0$$
$$\mathbf{r}_{cpu2} - \mathbf{m}_{p_3,r_{cpu2}} \geq 0$$

These first constraints cover the major requirements regarding process task binding and the allocation of the resources to execute them. The following constraints cover the aspect of routing data from sender to receiver and, thus, address Equations (7.3), (7.4), and (7.5):

$\forall c \in C, r \in R, (c, p) \in E_T, m = (p, r) \in E_M$:

$$\mathbf{c_r} - \mathbf{m} = 0 \tag{7.12}$$

$\forall c \in C$:

$$\sum_{r \in R} \mathbf{c_{r,1}} = 1 \tag{7.13}$$

$\forall c \in C, r \in R, (p, c) \in E_T, m = (p, r) \in E_M$:

$$\mathbf{m} - \mathbf{c_{r,1}} = 0 \tag{7.14}$$

Equation (7.12) ensures that a communication task c is routed to each resource a succeeding (receiving) process task is mapped to (cf. Equation (7.5)). Analogously, the constraints in Equations (7.13) and (7.14) ensure a communication task's root is the resource that the preceding (sending) process task is mapped to (cf. Equation (7.4)). Having the very basic routing constraints formulated, we need further constraints to precisely formulate what makes a route between source and multiple receivers feasible. First, we ensure that each communication task can only be routed on allocated resources by Equation (7.15):

$\forall c \in C, r \in R$:

$$\mathbf{r} - \mathbf{c_r} \geq 0 \tag{7.15}$$

Additionally, Equation (7.16) ensures that a communication task may be routed only between adjacent resources in one communication step:

$\forall c \in C, r \in R, \tau = \{2, .., |\mathcal{T}|\}$:

$$\left(\sum_{\tilde{r} \in R, e=(\tilde{r},r) \in E_R} \mathbf{c_{\tilde{r},\tau}} \right) - \mathbf{c_{r,\tau+1}} \geq 0 \tag{7.16}$$

It is finally required that a communication task is assigned a communication step τ if it is considered to be routed over a resource which is achieved by Equations (7.17) and (7.18):

$\forall c \in C, r \in R$:

$$\left(\sum_{\tau \in \mathscr{T}} \mathbf{c_{r,\tau}} \right) - \mathbf{c_r} \geq 0 \tag{7.17}$$

$\forall c \in C, r \in R, \tau \in \mathscr{T}$:

$$\mathbf{c_r} - \mathbf{c_{r,\tau}} \geq 0 \tag{7.18}$$

Here, Equation (7.17) ensures that if $\mathbf{c_r}$ is set to 1, the sum of $\mathbf{c_{r,\tau}}$ variables is greater zero which means that if the message is routed on the resource, a respective time step has to be assigned. On the other hand, Equation (7.18) ensures that no $\mathbf{c_{r,\tau}}$ variable can be set to 1 unless $\mathbf{c_r}$ is set to 1 as well.

The introduced Equations (7.10), (7.11), (7.12), (7.13), (7.14), (7.15), (7.16), (7.17), and (7.18) are suitable to ensure a feasible allocation, binding, and routing and, thus, a feasible implementation in case scheduling is of no concern.

Further constraints may be added to enhance the obtained feasible implementations. First, a natural enhancement to a feasible route is to require it to be free of (redundant) cycles. The satisfaction of Equation (7.19) avoids cycles in a route by ensuring that a communication task can pass a resource at most once:
$\forall c \in C, r \in R$:

$$\sum_{\tau \in \mathscr{T}} \mathbf{c_{r,\tau}} \leq 1 \tag{7.19}$$

Second, an implementation benefits from not containing unused (redundant) resources as they might affect design objectives such as cost, area, or power consumption. To eliminate unused resources from the allocation, Equation (7.20) ensures that a resource is only allocated if at least one process or communication task is bound to or routed over it:
$\forall r \in R$:

$$\left(\sum_{c \in C \wedge r \in R} \mathbf{c_r} \right) + \left(\sum_{m=(p,r) \in E_M} \mathbf{m} \right) - \mathbf{r} \geq 0 \tag{7.20}$$

The pseudo-Boolean encoding presented so far covers the allocation, binding, and routing.

As outlined in the definition of the system-level synthesis problem, we finally want to support generic scheduling constraints by means of assigning priorities. Thus, we now introduce a pseudo-Boolean encoding for task and communication priorities; cf. Equations (7.7) and (7.8). Here, we again need to introduce Boolean variables:

$\mathbf{s}_{p,\tilde{p}}$
: A Boolean variable that indicates whether process task $p \in P$ has a higher priority than task $\tilde{p} \in P$ (1) or not (0).

$\mathbf{s}_{c,\tilde{c}}$
: A Boolean variable that indicates whether communication task $c \in C$ has a higher priority than task $\tilde{c} \in C$ (1) or not (0).

The following constraints ensure that priorities are assigned properly. We first define the priority assignment function for process tasks which assigns correct priorities within tasks bound to the same resource (cf. Equation (7.7)):
$\forall r \in R, (p,r), (\tilde{p},r) \in E_M, p \neq \tilde{p}$:

$$\mathbf{s}_{p,\tilde{p}} + \overline{\mathbf{s}_{\tilde{p},p}} = 1 \tag{7.21}$$

Equation (7.21) states that if process task p has a higher priority than task $\tilde{p}$ ($\mathbf{s}_{p,\tilde{p}} = 1$), it has to be ensured that task $\tilde{p}$ has a lower priority than task p ($\mathbf{s}_{\tilde{p},p} = 0$). Now, we also have to ensure transitivity, i.e., task p has a higher priority than task $\tilde{p}$ and task $\tilde{p}$ than task $\hat{p}$. It also has to hold that task p has higher priority than task $\hat{p}$ which is ensured by Equations (7.22) and (7.23):
$\forall r \in R, (p,r), (\tilde{p},r), (\hat{p},r) \in E_M, p \neq \tilde{p} \neq \hat{p}$:

$$\mathbf{s}_{p,\tilde{p}} + \mathbf{s}_{\tilde{p},\hat{p}} + \overline{\mathbf{s}_{p,\hat{p}}} \leq 2 \tag{7.22}$$

$$\overline{\mathbf{s}_{p,\tilde{p}}} + \overline{\mathbf{s}_{\tilde{p},\hat{p}}} + \mathbf{s}_{p,\hat{p}} \leq 2 \tag{7.23}$$

Exactly the same requirements as ensured by Equations (7.21), (7.22) and (7.23) are now imposed on the communication tasks as well. The only difference is that – since communication tasks might share multiple buses and other communication resources – we apply a global priority assignment; cf. Equation (7.8):
$\forall c, \tilde{c} \in C, c \neq \tilde{c}$:

$$\mathbf{s}_{c,\tilde{c}} + \overline{\mathbf{s}_{\tilde{c},c}} = 1 \tag{7.24}$$

$\forall c, \tilde{c}, \hat{c} \in C, c \neq \tilde{c} \neq \hat{c}$:

$$\mathbf{s}_{c,\tilde{c}} + \mathbf{s}_{\tilde{c},\hat{c}} + \overline{\mathbf{s}_{c,\hat{c}}} \leq 2 \tag{7.25}$$

$$\overline{\mathbf{s}_{c,\tilde{c}}} + \overline{\mathbf{s}_{\tilde{c},\hat{c}}} + \mathbf{s}_{c,\hat{c}} \leq 2 \tag{7.26}$$

With the above-given constraints, a unique priority assignment is achieved.

From the introduced encoding Ψ_ς, a simple *decode* function as shown in Fig. 7.10 can be defined that derives the concrete implementation $\omega = \text{decode}(\mathbf{x}) = (G_\alpha, G_\beta, \gamma, S)$ from the phase of the Boolean variables in $\mathbf{x}$. This is also depicted in the decoding step shown in Fig. 7.10.

As outlined, the search space of SAT decoding consists solely of the two vectors σ and ρ of the branching strategy. Given the mentioned rules to determine an

encoding Ψ_ς, the search space can be seamlessly derived by providing one entry in σ and ρ for each variable required for the encoding Ψ_ς. This completes the introduction of the basic ingredients of the SAT-decoding approach.

7.4 Satisfiability Modulo Theories During Decoding

The introduced SAT-decoding approach is capable of restricting the search space to feasible implementations with respect to given linear constraints (cf. Definition 1) only. However, in the area of hardware/software codesign, several objectives can – typically – not be linearized in a sound fashion. Two prominent examples are *timeliness* which requires to analyze the interference of process and communication tasks on shared resources and *reliability* which is a probabilistic and combinatorial problem itself that has to consider which combination of faults in tasks or resources results in the system to fail. As can be imagined, transforming such complex behaviors and interactions into a combination of linear constraints may come at significant over-approximations or even result in practically useless results. In this section, we therefore introduce the key idea how to also consider non-linear constraints, followed by a formal definition of the SMT decoding technique. Afterward, different schemes how to learn which solutions are infeasible with respect to a set of non-linear constraints are discussed.

7.4.1 SMT Decoding: The Key Idea

The solution to the problem of considering non-linear constraints is inspired by the concept of *Satisfiability Modulo Theories* (SMT); cf. [4]. Without aiming for a complete and thorough introduction of SMT, the basic idea of SMT relevant for hardware/software codesign is that it checks the satisfiability of a logical formula over one or more *background theories*. The concept of *SMT decoding* [24] is depicted in Fig. 7.11. We again use the pseudo-Boolean encoding Ψ_ς that considers a set of linear constraints as introduced in the last section. We now hand over an implementation ω delivered by the solver to one or several background theories, each of which decides whether the implementation is feasible (isFeasible(ω)) for a set of non-linear constraints as well. In the context of hardware/software codesign, such a background theory could, for example, be a formal timing analysis (cf. ▸ Chap. 23, "CPA: Compositional Performance Analysis") or a timing simulation (cf. ▸ Chap. 19, "Host-Compiled Simulation") that can decide whether the delay of an implementation meets a certain deadline. So in particular, we *couple* any external analysis technique for any interesting system constraint as a background theory to the introduced solver. In case the variable assignment does not fulfill those constraints that are checked by the background theory, the solver will be told to consider this variable assignment as infeasible (although it initially appeared to be feasible considering only the set of linear constraints). This way, the solver

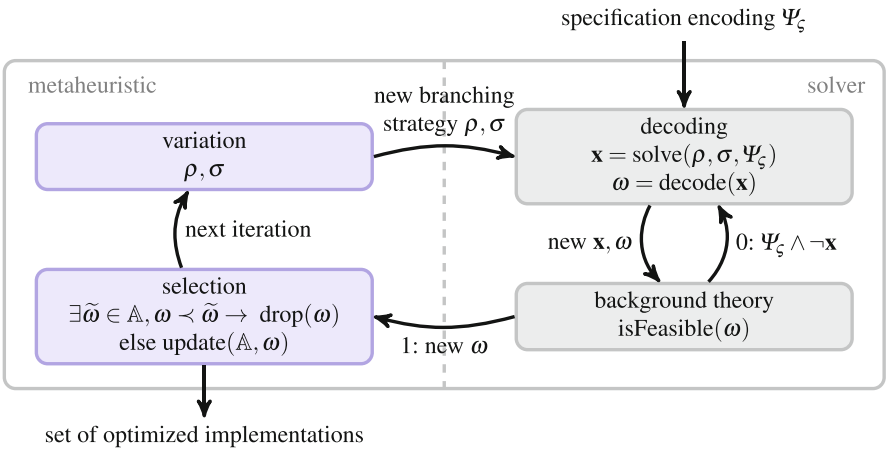


Fig. 7.11 Principle of the SMT decoding approach: The solver takes an encoding of the specification together with the current branching strategy and determines a feasible implementation ω with respect to a set of linear constraints. The background theory then checks the delivered implementation for feasibility with respect to a set of non-linear constraints. In case it is infeasible (0), the respective variable assignment is excluded in the solver and the solver is asked for a new implementation. This way, the solver *learns* over time which variable assignments violate a set of non-linear constraints

basically *learns* which variable assignments are infeasible with respect to a set of non-functional constraints over time.

Consider again the simple specification given in Fig. 7.1 and the implementation that is feasible with respect to the linear constraints defined by system-level synthesis depicted in Fig. 7.3 on the right. Assume we are interested in formulating a set of additional constraints on timeliness of computed results. For example, we formulate a deadline for the execution of the application of the simple system shown in Fig. 7.1. Here, we employ a formal timing analysis approach as our background theory to check whether the latency of each implementation meets the specified deadline. As indicated in Fig. 7.12 on the left, the implementation violates the deadline because p_2 and p_3 have to be executed sequentially on r_{cpu2} (depicted also in the Gantt chart at the bottom left). Thus, the implementation is infeasible with respect to timeliness and this information has to be propagated to the solver. This is achieved by combining the specification encoding Ψ_ς with an encoding $\mathbf{x}_\omega$ of this implementation such that this implementation is *not* feasible anymore. In the concrete example – and for the sake of brevity only considering the mapping variables – this results in $\Psi_\varsigma \wedge \neg(\mathbf{m}_{p_1,r_{cpu1}} \wedge \mathbf{m}_{p_2,r_{cpu2}} \wedge \mathbf{m}_{p_3,r_{cpu2}})$. This can be achieved by a Boolean conjunction of the specification encoding with the negated implementation encoding, i.e., $\Psi_\varsigma \wedge \neg \mathbf{x}_\omega$. Figure 7.12 on the right shows an implementation that adheres to both linear constraints and the non-linear constraint on timeliness – tasks p_2 and p_3 can be executed in parallel on r_{cpu2} and r_{cpu1}, respectively.

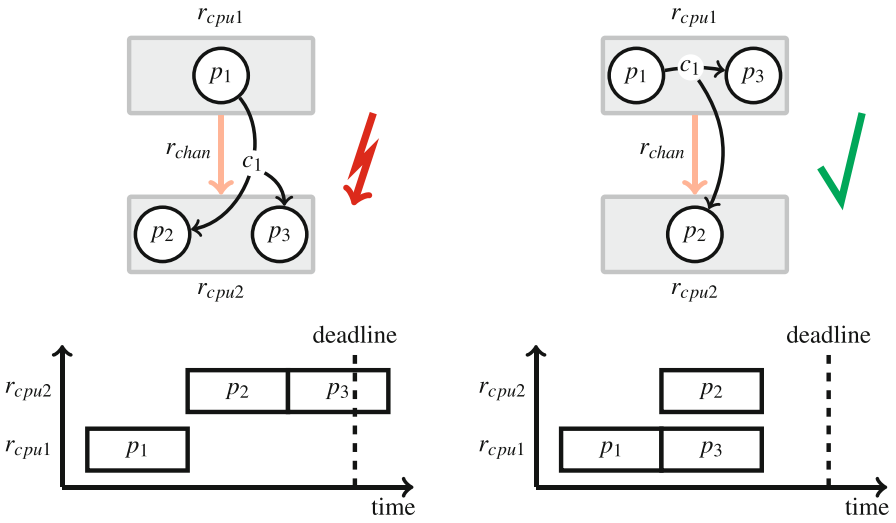


Fig. 7.12 Two implementations for the specification in Fig. 7.1 that are both feasible with respect to the introduced set of linear constraints. Yet, the implementation shown on *the left* does not meet a specified deadline because p_2 and p_3 have to run sequentially on r_{cpu2} after receiving the data from p_1 (see Gantt chart at *bottom left* obtained from the timing analysis). Thus, the implementation is infeasible with respect to a constraint on timeliness. On *the right*, the deadline can be met because p_2 and p_3 can run in parallel once both received the data from p_1 (see Gantt chart at *bottom right*)

This way, the key idea of SAT decoding – the restriction of the search space to feasible implementations only – can be extended to non-linear constraints as well by means of SMT decoding. In the following subsection, we give a formal definition of SMT decoding that completes the informal introduction given so far. Afterward, we introduce at which points of the solving process a feasibility check by the background theory can be applied, resulting in different *learning schemes* of the solver.

7.4.2 SMT Decoding Formulation

Let $\Omega_\mathrm{f} \subseteq \Omega$ denote the subset of feasible implementations of all implementations Ω. Those feasible implementations $\Omega_\mathrm{f} = \Omega_L \cap \Omega_N$ are given by the cut set of those implementations Ω_L that are feasible with respect to the set of linear constraints and implementations Ω_N that are feasible with respect to non-linear constraints. What we know from the previous section is that we can derive a pseudo-Boolean encoding Ψ_ς for our system-level synthesis problem that delivers Ω_L. Our aim is to derive an encoding Ψ_f for all feasible implementations Ω_f which would be given as $\Psi_\mathrm{f} = \Psi_\varsigma \wedge \Psi_N$. However, Ω_N cannot be converted to a respective Pseudo-Boolean (PB) encoding Ψ_N because we cannot linearize those constraints in a sound fashion. From this problem, we can formalize the key idea of SMT decoding:

In SMT decoding, an encoding Ψ_N for the set of implementations Ω_N that are feasible with respect to a set of non-linear constraints is derived by iteratively *learning* the implementations $\overline{\Psi_N}$ that are *infeasible* using one or several background theories. Whenever a variable assignment $\mathbf{x}$ is considered infeasible by the background theory, it is added to $\overline{\Psi_N}$ via $\overline{\Psi_N}^{i+1} := \overline{\Psi_N}^{i} \vee \mathbf{x}$. SMT decoding is, thus, capable of deriving Ψ_f via $\Psi_\mathrm{f} = \Psi_\varsigma \wedge \neg\overline{\Psi_N}$, i.e., the conjunction of those implementations that are feasible with respect to a set of linear constraints and *not* those that do violate any non-linear constraint.

Since SMT decoding aims at learning $\overline{\Psi_N}$ iteratively and does not require a closed-form representation, any analysis technique can be employed to determine whether an implementation is feasible or not. This results in a great flexibility and applicability of SMT decoding to various aspects and problems from the area of hardware/software codesign.

Note that this technique is even capable of covering a delicate corner case: In case no feasible implementation exists, i.e., $\Omega_\mathrm{f} = \emptyset$, the SMT decoding will iteratively eliminate infeasible implementations until the pseudo-Boolean solver returns a contradiction. At this moment, SMT decoding has proven that no feasible implementation exists which is neither possible for DSE approaches that solely rely on metaheuristic optimization nor for exact approaches that may only consider linear constraints.

7.4.3 Learning Schemes

For SMT decoding, three learning schemes have been proposed in literature: *Simple learning* requires no problem-specific knowledge while *early learning* requires that the specification allows to also derive *partial implementations*; see [24]. A third scheme that relies on the *deduction of justifications* requires that enough problem-specific knowledge is available such that the background theory can basically derive the *reason* why an implementation is infeasible; see [26]. In the following subsection, all three schemes are introduced.

7.4.3.1 Simple Learning

The simple learning scheme has already implicitly been mentioned in the introduction of SMT decoding. It is a direct implementation of the SMT decoding idea: The solver derives a variable assignment $\mathbf{x}$ which is passed to the background theories. If any of these recognizes that the respective implementation is infeasible, the variable assignment is added to the set of infeasible implementations $\overline{\Psi_N}$, i.e., $\overline{\Psi_N} := \overline{\Psi_N} \vee \mathbf{x}$.

Simple learning is depicted in Fig. 7.13: The triangle shall visualize the decision tree of the solver which is given by the Boolean variables and their phases. One path in that tree is one concrete variable assignment $\mathbf{x}$ and the solver will, of course, only consider those variable assignments that are feasible with respect to the set of linear

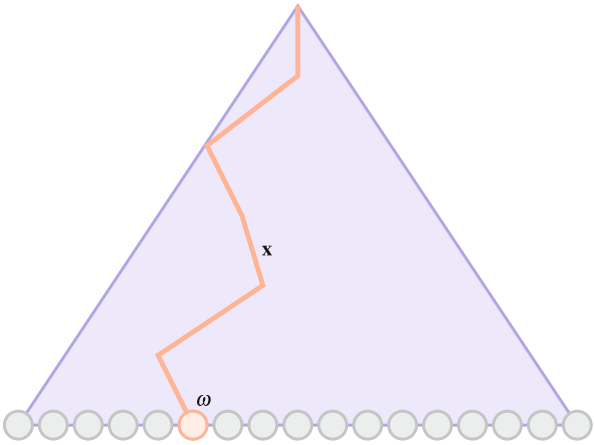


Fig. 7.13 The decision tree that is given from the encoding Ψ_ς. The leaves of the tree denote the set of implementations Ω_L that are feasible with respect to a given set of linear constraints. In the simple learning scheme, each complete variable assignment $\mathbf{x}$ or implementation ω, respectively, is checked for feasibility by the background theories. Thus, the simple learning scheme can only eliminate individual implementations in case of infeasibility

constraints. The leaves of the tree are all variable assignments or points in our search space that are feasible with respect to the set of linear constraints. As can be seen, simple learning considers each point in the search space individually and checks its feasibility; possibly forbidding it for future solving by means of learning.

The advantage of the simple learning is that the background theories can be treated as black-box analysis approaches. The simple learning delivers complete implementations, asks for feasibility using the background theories, and – if required – eliminates a complete implementation from the search space. The drawback is that, for large search spaces, many very similar implementations might exist that all violate certain non-linear constraints. With simple learning, those would have to be checked individually which might be computationally expensive. In the worst case, no feasible implementation might exist such that simple learning becomes an exhaustive search. Note that not only the sheer number of checked implementations might become a problem, but also the huge number of PB constraints that are iteratively added to $\overline{\Psi_N}$ which may become a problem for efficiently solving of the resulting function Ψ_f.

7.4.3.2 Early Learning

The early learning scheme tries to overcome the outlined problems of the simple learning scheme by trying to evaluate already *partial implementations* [24]. The idea of early learning is depicted in Fig. 7.14. There, already a *partial variable assignment* $\mathbf{x}'$ which corresponds to a partial implementation ω' is checked by the background theories for feasibility. The significant advantage arises in case such a partial variable assignment is infeasible: Not only one implementation, but all complete implementations that are based on the partial implementation can be eliminated from the search space at once by $\overline{\Psi_N} := \overline{\Psi_N} \vee \mathbf{x}'$.

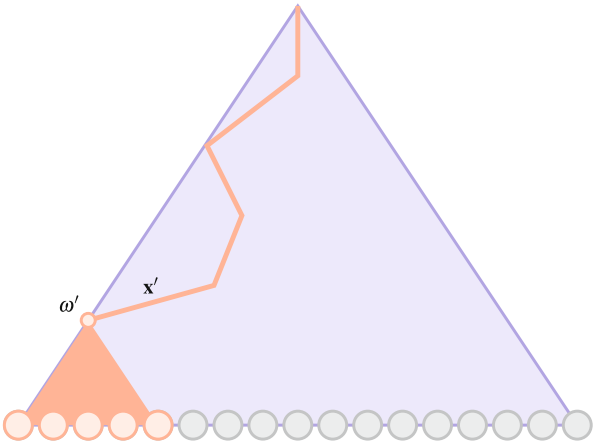


Fig. 7.14 Depicted is again the decision tree given by the encoding Ψ_S. In the early learning scheme, already a partial variable assignment $\mathbf{x}'$ that represents a partial implementation ω' is checked for feasibility by the background theories. Given the partial implementation is already infeasible, a complete subtree of the decision tree can be eliminated, eliminating several – in the concrete example five – implementations at once

However, opposed to the simple learning scheme, care must be taken when applying the early learning scheme.

For early learning, it has to hold that in case a partial implementation is infeasible, all implementations that contain this partial implementation must be infeasible as well. This holds true if the background theory is *monotonic* with respect to partial implementations.

Of course, whether this assumption holds or not heavily depends on the used background theory: Consider, for example, our previous constraint on timeliness of an implementation and a schedule analysis used as background theory. If already a partial implementation violates a given deadline, it will typically hold that a complete implementation with more workload and/or interference in the system will also violate the deadline. For many timing analysis approaches, early learning can be used. On the other hand, a consideration of system reliability may result in a different situation. While a partial implementation might not satisfy a constraint on minimal lifetime, a complete implementation might add additional redundant resources or tasks to the system. With this redundancy, the lifetime criterion might again be met by the complete implementation. But, as discussed in [24], a clever and problem-specific variable ordering might allow to employ early learning at *safe points* in the decision tree such that monotonicity of the background theory is achieved.

7.4.3.3 Deducing Justifications

We recognized that the early learning scheme already offers several advantages over the simple learning strategy but requires a monotonic background theory by deriving partial implementations at *safe points* during the run of the solver. This could be avoided by the simple learning scheme which, however, comes at the drawback of only being able to eliminate one implementation per check. A third learning scheme that explicitly targets this problem is based on the following idea:

> The violation of a certain non-linear constraint is typically not caused by *all* assigned decision variables, but only by a subset of critical decisions termed *justification*.

Consider again the example of a background theory that analyzes timeliness. The violation of a deadline is typically caused by the critical path in the implementation. However, not all design decisions contribute to the critical path, but only a subset of design decisions that cause interference on computation and communication resources. The key idea of this learning scheme is to rely on background theories that consider an implementation ω and the respective variable assignment $\mathbf{x}$, check its feasibility, and deliver the justification $\widetilde{\mathbf{x}}$. Eliminating the justification from the search space has the immediate effect that not only one – as in early learning – but multiple complete subtrees can be removed from the decision tree or search space, respectively. In particular, it eliminates all implementations that contain the determined justification or, in other words, that include the critical decisions that will always result in a violation of the constraint. The concept of this learning scheme is depicted in Fig. 7.15.

Similar to the early learning scheme, it has to hold that all implementations that contain the justification do violate the respective constraint. However, opposed to ensuring this via a respective variable ordering and interrupting the solver, this learning scheme only relies on the simple learning considering the solver while it is the task of the background theory alone to determine the justification. Thus, it can be concluded that the deduction of justifications can be considered the least invasive and most flexible learning approach, given a respective background theory is available.

7.5 Applications

The introduced techniques of SAT and SMT decoding may be employed to a variety of constrained combinatorial problems, of which several are highly relevant for hardware/software codesign at system level. In this section, we will briefly outline some concrete applications of the introduced techniques to serve as directions for further reading and to give evidence of the flexibility and applicability of the underlying ideas.

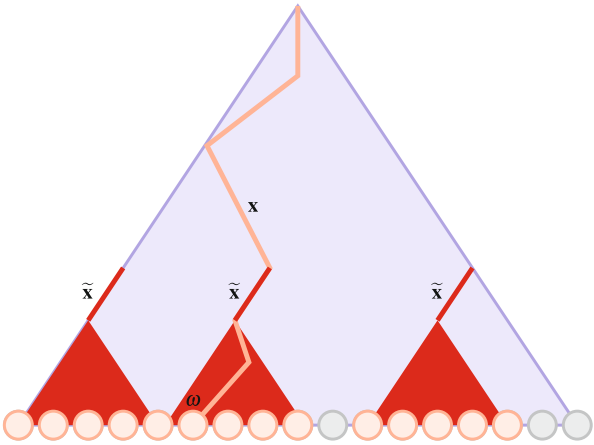


Fig. 7.15 Depicted is again the decision tree given by the encoding Ψ_ς. As in the simple learning scheme, a variable assignment $\mathbf{x}$ that represents an implementation ω is checked for feasibility by the background theories. But here, the background theory has the capability to derive the set of variables and their phase $\widetilde{\mathbf{x}}$ called justification that causes the violation of non-linear constraints. Learning this justification, not only one, but possibly multiple complete subtrees of the decision tree can be eliminated at once

SAT decoding is successfully applied to system-level synthesis problems from the area of Multi-Processor System-on-Chip (MPSoC) design; see, for example, [21]. There, the focus is on the distribution of process tasks to multiple processing units as well as hardware accelerators and also to find a specification encoding that suits the application's Model of Computation (MoC); see ▸ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign". For upcoming many-core architectures that often feature regular communication topologies such as meshes, SAT decoding is extended to mitigate the complexity increase of the routing in such architectures; see [10]. For many-core architectures and so-called hybrid mapping approaches (see ▸ Chap. 10, "Design Space Exploration and Run-Time Adaptation for Multicore Resource Management Under Performance and Power Constraints"), SAT decoding is used as part of the design-time DSE [31].

A particular domain where the concepts of SAT and SMT decoding are applied is networked embedded systems as can be found in avionics, rail, industrial automation, and automotive systems. Here, routing data over multiple different field bus systems is one problem where SAT decoding enables a conclusive solution [17], particularly for automotive Electric and Electronic (E/E) architectures. Besides the integration of various applications, SAT decoding is also used to integrate additional features such as diagnosis applications [25] like Built-In Self-Tests (BISTs) that must not interfere with the applications yet enhance the quality of the system. SMT decoding is applied to automotive applications with stringent real-time requirements in [26]. The approach in [14] uses a concept similar to SMT decoding to design automotive systems that are completely time-triggered and combines architectural and timing optimization in a unified DSE.

SAT decoding has also been used for the design of dependable embedded systems where dependability-enhancing techniques such as the binding of redundant process or communication tasks are integrated directly into the specification encoding; see, for example, [7]. The application of SMT decoding to consider dependability constraints such as a minimal expected lifetime is discussed in [24].

Finally, modern embedded systems may not only implement a fixed set of applications but rather enable customers to select various features and, thus, create their individual *variant* of the system. Particularly in the automotive domain, *variant management* requires to keep track of both the variants arising from a combination of different applications and the underlying architecture that has to support the different application variants in an efficient fashion. The approaches in [9] and [8] target these problems using the SAT-decoding technique.

Availability of the techniques: The described SAT-decoding approach is publicly available at [18] as part of the open-source library OPT4J [19] which can serve as a base for the application of SAT decoding to a wide range of constraint combinatorial problems. An open-source library termed OPENDSE is also publicly available [20] which already combines the SAT-decoding engine of OPT4J with a system model suitable for system-level DSE and hardware/software codesign as introduced in this chapter.

7.6 Conclusion

This chapter introduces a hybrid optimization approach to be used during Design Space Exploration (DSE) for system-level hardware/software codesign. The targeted problem is that linear as well as non-linear constraints may render many system implementations infeasible, such that classic DSE approaches can hardly find high-quality implementations or – in extreme cases – cannot even find a single feasible system implementation. The main focus of this chapter is the introduction of a hybrid optimization approach that allows a metaheuristic optimization to derive feasible implementations using a nested exact technique. The first method termed SAT decoding is capable of considering linear constraints and is used to introduce the general concept of *hybrid optimization*. Since hardware/software codesign at system level typically also has to respect constraints that cannot be expressed as linear constraints such as on timeliness, power consumption, or reliability, a second approach is introduced that may also handle additional non-linear constraints. This approach termed SMT decoding is capable of employing any available analysis technique to judge whether an implementation violates a given set of non-linear constraints or not and, thus, learns which implementations are infeasible in an iterative but efficient way. Moreover, three different learning schemes are introduced that either require no problem-specific knowledge at all or can significantly improve the learning via the evaluation of partial implementations or the deduction of the cause of a constraint violation. The chapter is concluded with examples of the successful application of the SAT and SMT decoding approaches to different areas such as MPSoC design and automotive systems.

References

1. Blickle T, Teich J, Thiele L (1998) System-level synthesis using evolutionary algorithms. Des Autom Embed Syst 3(1):23–58
2. Coello Coello CA (2002) Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. Comput Methods Appl Mech Eng 191(11–12):1245–1287
3. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5(7):394–397
4. De Moura L, Bjørner N (2011) Satisfiability modulo theories: introduction and applications. Commun ACM 54(9):69–77
5. Gajski DD, Kuhn RH (1983) New VLSI tools. IEEE Comput 16(12):11–14
6. Gerstlauer A, Haubelt C, Pimentel A, Stefanov T, Gajski D, Teich J (2009) Electronic system-level synthesis methodologies. IEEE Trans Comput Aided Des Integr Circuits Syst 28(10):1517–1530
7. Glaß M, Lukasiewycz M, Reimann F, Haubelt C, Teich J (2010) Symbolic system level reliability analysis. In: Proceedings of the international conference on computer-aided design (ICCAD), San Jose, pp 185–189
8. Graf S, Glaß M, Teich J, Lauer C (2014) Multi-variant-based design space exploration for automotive embedded systems. In: Proceedings of design, automation and test in Europe (DATE), p 6
9. Graf S, Glaß M, Wintermann D, Teich J, Lauer C (2013) IVaM: implicit variant modeling and management for automotive embedded systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), p 10
10. Graf S, Reimann F, Glaß M, Teich J (2014) Towards scalable symbolic routing for multi-objective networked embedded system design and optimization. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 2:1–2:10
11. Hernandez-Aguirre A, Botello-Rionda S, Coello Coello CA, Lizarraga-Lizarraga G, Mezura-Montes E (2004) Handling constraints using multiobjective optimization concepts. Int J Numer Methods Eng 59(15):1989–2017
12. Kienhuis ACJ (1999) Design space exploration of stream-based dataflow architectures – methods and tools. Ph.D. thesis, Delft University of Technology
13. Le Berre D, Parrain A (2010) The Sat4J library, release 2.2. system description. J Satisf Boolean Model Comput 7:59–64
14. Lukasiewycz M, Chakraborty S (2012) Concurrent architecture and schedule optimization of time-triggered automotive systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 383–392
15. Lukasiewycz M, Glaß M, Haubelt C, Teich J (2007) Solving multiobjective Pseudo-Boolean problems. In: Proceedings of the international conference on theory and applications of satisfiability testing (SAT), pp 56–69
16. Lukasiewycz M, Glaß M, Haubelt C, Teich J (2008) Efficient symbolic multi–objective design space exploration. In: Proceedings of the Asia and South Pacific design automation conference (ASPDAC), Seoul, pp 691–696
17. Lukasiewycz M, Glaß M, Haubelt C, Teich J, Regler R, Lang B (2008) Concurrent topology and routing optimization in automotive network integration. In: Proceedings of the design automation conference (DAC), Anaheim, pp 626–629
18. Lukasiewycz M, Glaß M, Reimann F Opt4J–meta-heuristic optimization framework for java. http://www.opt4j.org/
19. Lukasiewycz M, Glaß M, Reimann F, Teich J (2011) Opt4J: a modular framework for meta-heuristic optimization. In: Proceedings of the genetic and evolutionary computation conference (GECCO), pp 1723–1730
20. Lukasiewycz M, Reimann F OpenDSE–open design space exploration framework. http://opendse.sourceforge.net/

21. Lukasiewycz M, Streubühr M, Glaß M, Haubelt C, Teich J (2009) Combined system synthesis and communication architecture exploration for MPSoCs. In: Proceedings of design, automation and test in Europe (DATE), pp 472–477
22. Prakash S, Parker AC (1992) SOS: synthesis of application-specific heterogeneous multiprocessor systems. J Parallel Distrib Comput 16(4):338–351
23. Puchinger J, Raidl G (2005) Combining metaheuristics and exact algorithms in combinatorial optimization: a survey and classification. In: Proceedings of the first international work-conference on the interplay between natural and artificial computation (IWINAC), vol 3562, pp 41–53
24. Reimann F, Glaß M, Haubelt C, Eberl M, Teich J (2010) Improving platform-based system synthesis by satisfiability modulo theories solving. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 135–144
25. Reimann F, Glaß M, Teich J, Cook A, Gómez LR, Ull D, Wunderlich HJ, Abelein U, Engelke P (2014) Advanced diagnosis: SBST and BIST integration in automotive E/E architectures. In: Proceedings of the design automation conference (DAC), p 8
26. Reimann F, Lukasiewycz M, Glaß M, Haubelt C, Teich J (2011) Symbolic system synthesis in the presence of stringent real-time constraints. In: Proceedings of the design automation conference (DAC), pp 393–398
27. Smith AE, Coit DW (1997) Penalty functions, chap. C 5.2. Institute of Physics Publishing and Oxford University Press, Bristol
28. Teich J (2012) Hardware/software co-design: past, present, and predicting the future. Proc IEEE 100(5):1411–1430
29. Teich J, Blickle T, Thiele L (1997) An evolutionary approach to system-level synthesis. In: Proceedings of the international workshop on hardware/software codesign (CODES/CASHE), pp 167–171
30. Teich J, Haubelt C (2007) Digitale hardware/software-systeme: synthese und optimierung, 2nd edn. Springer, Heidelberg
31. Weichslgartner A, Gangadharan D, Wildermann S, Glaß M, Teich J (2014) DAARM: design-time application analysis and run-time mapping for predictable execution in many-core systems. In: Proceedings of the international conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 34:1–34:10
32. Zitzler E, Thiele L (1999) Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. IEEE Trans Evol Comput 3(4):257–271