Exploiting Similarity in Evolutionary Product Design for Improved Design Space Exploration *

Luise Müller[0000−0002−3924−5852], Kai Neubauer[0000−0003−4138−5114], and Christian Haubelt[0000−0002−1568−5423]

Applied Microelectronics and Computer Engineering, University of Rostock, Germany
{luise.mueller,kai.neubauer,christian.haubelt}@uni-rostock.de

Abstract. The design of new products is often an evolutionary process, where product versions are built on one another. This form of *product generation engineering (PGE)* reuses some parts of previously developed systems, while others have to be designed from scratch. In consideration of subsequent design steps, i.e., verification, testing, and production, PGE may significantly reduce the time-to-market as these steps can be skipped for reused parts. Thus, deciding which components have to be replaced or added to meet the updated requirements while preserving as many legacy components as possible is one of the key problems in PGE. A further aspect of PGE is the potentially more efficient search for valid design candidates. An already optimized base system can be systematically extended by new functionality without the necessity to search the entire design space. To this end, in this work, we propose a systematic approach, based on Answer Set Programming, to exploit the ideas of PGE in electronic system-level design space exploration. The idea is to gather information on a previous design, analyze the changes to a new version, and utilize the information to steer the search towards potentially good regions in the design space. Extensive experiments show that the presented approach is capable of finding near-optimal design points up to 1,000 times faster than a conventional approach.

Keywords: Design Space Exploration · Heuristic · Answer Set Programming · Evolutionary Design

1 Introduction

Embedded computer systems continuously advance into more areas of everyday life such as medical devices, automotive industry, and telecommunications. In addition to the growing number of application areas, the complexity of individual systems, influenced by the number of internal components, processes, and heterogeneity, grow simultaneously. Due to the growing complexity, for each system, a vast amount of design decisions has to be made that influence the characteristics of the system. This includes the allocation of hardware resources,

* This work was funded by the German Science Foundation (DFG) under grant HA 4463/4-2

the partitioning of functionalities into hardware and software, and the synthesis of the communication infrastructure. The aim is not only to design a valid system, but also to optimize the resulting characteristics of a product. Furthermore, stringent time-to-market requirements, imposed by the pace of technological progress, aggravate the problem of designing optimal products. Hence, an efficient design space exploration is imperative to deliver high-quality products in a reasonable timeframe. To this end, the design process is started at high abstractions with lower degrees of detail mitigating complexity and allowing for a quick exploration of promising design points. Although the DSE is started at a high abstraction level to accelerate decision-making, the high complexity of today's computer systems prevents a complete exploration of the search space. Hence, finding optimal design points remains complicated.

In reality, many electronic systems do not have to be developed entirely from scratch. Instead, it is often aspired to have an entire product line with multiple variants of the system as well as potential successor devices where only marginal changes to the specification are made. Thus, the development of systems can be recognized as an evolutionary process where product versions are built on one another. This form of product design, called *product generation engineering* (PGE) [1], reuses components of previous versions, while others have to be designed from scratch. Deriving a version of an existing product can mitigate the design time and limits verification and testing to the new parts of the system.

Assume, for example, the product line of current smartphone manufacturers. Regularly, typically in a one-year interval, a new generation is released. Here, a generation consists of a base device and derivatives that either have specialized camera sub-systems, less processing capabilities, or varying display and battery sizes. While the transition from one generation to the next may be larger than the changes within one generation, core parts, such as the wireless interface (e.g., WiFi, Bluetooth, GSM) or parts of the operating system remain subject to reuse.

To exploit the general trend towards PGE, in this paper, we propose a methodology that detects similarities of systems in an evolutionary design process. The obtained information is subsequently used in the design process aiming to keep implementation decisions. Our contribution is threefold:

1. We provide an extension to a state-of-the-art system-level design space exploration framework. The information of design decisions of previous product versions is extracted and used to steer the search towards promising regions in the design space exploration of the new product version.
2. We propose a declarative encoding of the problem through the utilization of Answer Set Programming (ASP). This results in a succinct and elaboration tolerant formulation that is easily extensible for future problems.
3. An extensive study is executed that evaluates the proposed approaches with a varying number of changes. The results indicate a large improvement on the quality of found solutions when compared to traditional approaches where no information of previous generations can be used. While the overall exploration time is not reduced with the presented approach, the exploration yields good solutions three orders of magnitude faster on average.

2 Related Work

In previous works, an effective design reuse model has been developed in [3] and the question of reusability addressed in [6]. Therefore, the necessity of knowledge reuse in connection with product design tasks has been discovered already decades ago. Nevertheless, the advantages of design reuse, like time savings, the prevention of faults as well as an increased extensibility and predictability [3, 6] are still valid nowadays and are targeted by the approach of this paper. To make a design applicable for reuse, steps, such as documentation, standardization, parameterization and modularization are carried out [3] enabling that the concept of design reuse is used in processes like design exchange, design evolution or component-based design [3, 6].

As an example, the composition of existing subsystems can be implemented by the use of hierarchical mapping encodings, which represent the assignment of functionality to architectural resources [10]. While in that concept subsystems are modeled and combined during the system synthesis steps, the proposed approach aims at identifying similarities between two product versions to reuse parts from one another on design decision level. The synthesizing problem can successfully be encoded using SAT [10] or answer set programming (ASP) [2, 11, 12]. Contrary to SAT, ASP is based on a closed-world assumption which allows an efficient implementation in particular for densely connected networks and multi-hop communication [2, 12].

As another application, the concept of PGE combines reuse mechanisms with significant new developments during the generation of a new technical product. That allows to build up generations of products based on a reference product. Such product management expands the view by an economic perspective. Concrete technical use cases are illustrated by the product generations of the Porsche 911 or of the iPhone [1]. A platform-based design, moreover, enables the creation of either module-based or scale-based product families. For those, metrics and optimization algorithms have been classified [15].

To be able to make use of prior design decisions, similarities and differences between two product versions have to be identified. In this approach, the components of a specification graph are considered whereas in [4] a similarity analysis and scoring is performed on call graphs from different control software projects. In another approach, equivalent mappings for symmetrical transformations of the architecture are determined, thus reducing the number of feasible solutions [7].

3 Fundamentals

In this section, we will give an overview of the key prerequisites for the remainder of the paper. To this end, we first present the underlying system model used throughout the paper. Subsequently, the exploration approach is defined, that includes the concept of Pareto optimality and the synthesis model used. Finally, we introduce Answer Set Programming as the symbolic solving technology that is employed to realize the concept.

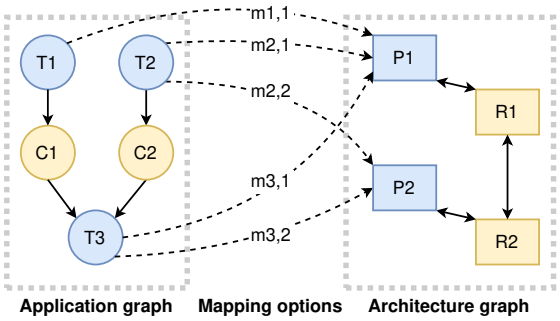


Fig. 1: An example of a specification graph

3.1 System Model

In this paper, we specify the system at the electronic system level (ESL). The specification $S = (A, H, M)$ is split into an application, modeling the behavior of the system, and a hardware template constraining the structure of the system. Both the application $A = (V_A, E_A)$ and the hardware template $H = (V_H, E_H)$ are modeled through directed graphs and are connected through a set of mapping options M, as depicted in Fig. 1. The application is modeled at a task-level granularity with the set of vertices $V_A = T \cup C$ consisting of computational tasks T and communication messages C. The edges $E_A \subseteq T \times C \cup C \times T$ represent the data flow of the application and therefore, the interdependencies of individual tasks. Tasks can send and receive messages to exchange data packets according to their behavior. Each message $c \in C$ is required to be sent and received exactly once, i.e., $\nexists c : \{(c, t_i), (c, t_j)\} \subset E_A$ and $\nexists c : \{(t_i, c), (t_j, c)\} \subset E_A$. In other words, only point-to-point communication among tasks can be modeled directly. Hence, multicast communication is modeled through multiple messages that are all sent by the same task but have different receivers.

The vertices of the hardware template $V_H = P \cup R$ represent hardware devices and are separated into processing elements P and routing units R. While the processing elements are used to execute the tasks of the application graph, the routing units cannot execute code but rather form the communication infrastructure. The latter is completed by the edges $E_H \subseteq V_H \times V_H$ representing links that establish communication channels between devices. In contrast to the application graph, the links are not constrained, i.e., potentially each device may be connected to another device through a link. In this work, we focus on networks on chip (NoC) with regular mesh topologies. However, in principle, the same approach can be used to model bus-based or mixed hardware architectures. Note that the bidirectional edges in Fig. 1 represent two individual links. For example, the edge between routers R1 and R2 is modeled through the two links $l_1 = (R_1, R_2)$ and $l_2 = (R_2, R_1)$.

The set of mapping options $M \subseteq T \times P$ connects the application and hardware graphs. At least one mapping option $m \in M = (t_i, p_j)$ is defined for each

task that signifies that the task t_i may be executed on p_j. For the messages, no mapping options have to be specified explicitly as they are constrained implicitly by their sending and receiving tasks, respectively, and can be routed over the entire communication infrastructure. The function $w : M \to \mathbb{N}$ assigns an integer number to each mapping option $m = (t, p)$, signifying the worst case execution time of the task t on the processing element p. Analogously, further properties are assigned to the remaining elements of the specification graph to model heterogeneous architectures. In the present paper, we define the functions $P_{stat} : V_H \to \mathbb{N}$, $area : V_H \to \mathbb{N}$, and $E_{dyn} : M \to \mathbb{N}$ that assign the static power consumption and area costs to each hardware device as well as the dynamic energy requirements to each mapping option, respectively. Finally, a periodicity $\mathcal{P}$ and a routing energy E_r are assigned to the specification that specifies the time, after which the execution is restarted and the energy a single message hop consumes when routed over the network, respectively[1].

In order to transform the specification into an implementation, a valid allocation, binding, routing, and schedule have to be determined. The allocation α is composed of devices and links from the heterogeneous architecture template H, i.e., $\alpha \subseteq V_H \cup E_H$ that shall be used in the specific system implementation and is separated into the device and link allocation α_D and α_L. The static binding $\beta \subseteq M$ and routing $\gamma \subset C \times 2^{E_H}$ select exactly one mapping option for each task and a cycle-free route for each message, depending on the binding of the sending and receiving tasks, respectively. Finally, the schedule τ assigns start times to each task and message, i.e., $\tau : T \cup C \mapsto \mathbb{N}$.

3.2 Exploration Model

The aim of the design space exploration (DSE) is the determination of a set of Pareto-optimal implementations of a specification $S = (A, H, M)$. To this end, each implementation x has to be evaluated according to a set of desired objective functions. In this paper, we focus on the overall latency $lat(x)$ of the system, its area costs $area(x)$, and the energy requirements $E(x)$. Without loss of generality, the DSE is formulated as a multi-objective minimization problem:

$$\begin{aligned} &\textbf{minimize } f(x) = (lat(x), area(x), E(x)), \\ &\text{subject to:} \\ &\quad x \text{ is a feasible system implementation.} \end{aligned}$$

The area costs of an implementation are calculated as the accumulated area costs of each allocated hardware device, i.e., $area(x) = \sum_{d \in \alpha_D} area(d)$. The energy requirement is the sum of the systems dynamic and static energy requirements:

$$E(x) = \mathcal{P} \cdot \sum_{d \in \alpha_D} P_{stat}(d) + \sum_{m \in \beta} E_{dyn}(m) + \sum_{r \in \gamma} E_r \cdot hops(r).$$

[1] For simplicity, we restrict the properties to integer values. The proposed ASPmT-based [5] approach, however, also allows for real-valued properties in principle.

Note that we refer the static energy to one iteration of the system, i.e., the consecutive execution of all tasks within the given period $\mathcal{P}$. The latency of the system is defined as the difference between the maximum end time ($\tau(t) + w((t,p))$, i.e., depending on β) and the minimum start time ($\tau(t)$):

$$lat(x) = \max_{(t,p)\in\beta} (\tau(t) + w((t,p))) - \min_{t\in T}(\tau(t)).$$

For the sake of brevity, we will forgo the exact details of the evaluation steps as they are not particularly relevant for the proposed approach at hand. Instead, we refer to [11] for further information.

As is common in multi-objective optimization problems with conflicting objectives f_i, a single optimal solution generally does not exist as solutions are not totally, but only partially ordered through the dominance relation $\succ$. The dominance relation $\succ$ is defined for n-dimensional quality vectors of two distinct solutions. A candidate solution x dominates another solution y ($x \succ y$) if x evaluates at least as good in every objective and better in at least one objective compared to y. Without loss of generality, for a minimization problem with n objectives, it is formally defined as follows:

$$x \succ y \leftrightarrow \forall i \in \{1,\ldots,n\} : f_i(x) \leq f_i(y) \ \wedge \exists j \in \{1,\ldots,n\} : f_j(x) < f_j(y). \quad (1)$$

A solution x is said to be Pareto-optimal if no dominating solution y exists. Hence, by definition, Pareto-optimal solutions in the Pareto set X_P for a given problem are mutually non-dominated to each other: $\nexists x, y \in X_P : x \succ y \vee y \succ x$.

3.3 Answer Set Programming

In the paper at hand, we implement the DSE with ASP, a programming paradigm that stems from the area of knowledge representation and reasoning. In the following, we will introduce the basics of ASP that are imperative to understand the core concepts of the present work. Based on the stable model semantics, ASP is tailored towards NP-hard search problems. The input is a logic program formulated in a first-order language that is typically separated into a general problem description and a specific problem instance. While the former consists of rules that define how new knowledge is inferred, the latter contains facts representing the initial knowledge. A stable model, or *answer set*, of a logic program conforms to a feasible variable assignment that can be inferred by the rules applied to the facts. The knowledge is encoded by n-ary predicates, i.e., *atoms*, consisting of a predicate name and n parameters. For example, the unary atom `task(ti)` may encode the existence of a task $t_i \in T$, while the binary atom `map(t,p)` indicates that task t may be executed on the processing element p.

An ASP rule consists of a head and a body, and indicates that its head can be inferred if the body holds. In its simplest form, a rule has an empty body and therefore holds unconditionally, i.e., represent the facts to model initial knowledge. In contrast, a rule with an empty head, called an integrity constraint, forces the body not to hold. This way, specific assignments can be excluded from a stable model. To allow for the general problem description to

be applicable to each problem instance, the rules are encoded with variables, generally indicated by uppercase letters in the encoding. For instance, the rule `1{bind(T,P) : map(T,P)}1 :- task(T).` encodes the binding constraint. The rule states that exactly one mapping option $(t, p) \in M$ has to be selected for each task $t \in T$. Internally, the rule is grounded into a variable-free representation resulting in a set of $|T|$ individual rules. Afterwards, the variable-free atoms are inferred according to the rules given by the problem definition. Therefore, the ASP solver employs a conflict-driven clause learning (CDCL) strategy where atoms are inferred subsequently until a conflicting assignment causes the generation of a conflict clause and the back-jump to a previous decision level.

The order, in which (variable-free) atoms are assigned, is decided by a heuristic that is generally influenced by a generic set of rules, the characteristics of the problem. These rules can be disparate and usually influence the performance of the search differently for varying problem classes. The utilized ASP solver clingo, for example, employs the heuristic Variable State Independent Decaying Sum (VSIDS) [8] in its default configuration. Here, variables are assigned initial activities that decay over time and increase if they appear in a learned conflict clause. Whenever the search branches, the solver selects the atom with the highest activity. Although VSIDS is considered to be one of the most efficient branching heuristics [8], it does not embody domain knowledge. In the paper at hand, we will propose the use of domain specific heuristics to accelerate the evolutionary product design. This is discussed in more detail in Sec. 4.3.

Note that an elaborate discussion of ASP solving and the detailed presentation of the encoding are out of the scope of this paper. The interested reader is referred to [9, 5] and [12], respectively.

4 Similarity of Design Points

The aim of this project is to enhance the development step of the DSE by applying the idea of evolutionary product design. The therefore required prior knowledge is provided by a previously developed system representing a product present on the market. It is given by the parent configuration in Fig. 2 and is consisting of a specification and an implementation. It has to be noted that this solution is not guaranteed to be optimal, but very good concerning its application. Besides, Fig. 2 illustrates the steps of the proposed approach.

As a comprehensive example, a cellphone shall be improved. By modifying the specification of the parent configuration, a new derived version is created, namely the child configuration. Since all elements of the specification offer modification potential, it might be planned to extend the device functionality by changing its application as well as to equip the cell phone with additional hardware components, like a new processor or a second camera to improve its performance. Depending on the extent of the modification and the affected sections, a change in the specification can have a considerably large as well as nearly no impact on the final implementation. To refer to the example given above, an additional processor is only allocated, when a task is bound on it. If done so, addi-

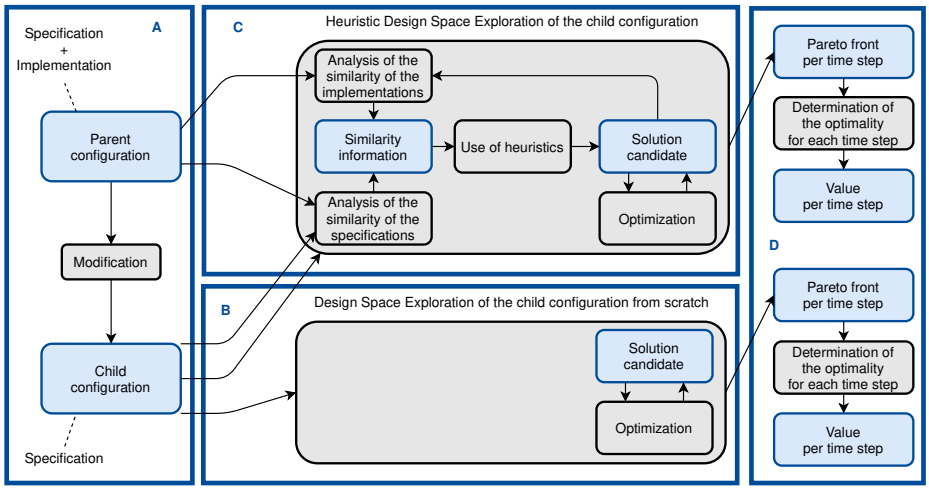


Fig. 2: Overview of the proposed approach (A+C) and its comparison (D) to a state-of-art approach (B)

tional interconnections are required to ensure the communication ability of the new processor. The modification stage is shown in block A in Fig. 2. The specific modifications applied in the experiments are given in section 5.

To determine the implementation of the child configuration, two approaches are distinguished. Block B in Fig. 2 represents the DSE from scratch where solution candidates are generated. Only valid intermediate solutions are kept and further optimized concerning the factors latency, energy consumption and area costs. This procedure is enhanced by the use of heuristics during the DSE shown in block C in Fig. 2. The heuristic DSE aims at retaining as many design decisions from the parent implementation as possible. To gain knowledge about reusable concepts and design decisions from the parent configuration, the specifications of both systems are analyzed to identify the similarities and differences and thus, the reuse potential. The corresponding steps are explained in detail in the following subsections.

Finally in block D, the results of both approaches are compared. Therefore, for each the optimality of the Pareto front of the final and of all intermediate solutions is evaluated and set in context to the time when the individual solution has been found.

4.1 Analysis of the Specifications

Firstly, as shown in Fig. 2, all elements of the specification graph as well as all characteristics of both configurations are examined and compared to identify each similarity and difference. From the perspective of the child configuration, for each instance of each component type, it is recorded whether it is a common (*equal*), an own, i.e., newly *added*, or an unknown and therefore already deleted

(*missing*) component. These three cases are demonstrated by means of the component type *task* in Code snippet 1. The comparison is carried out on the basis of the parameters of the component instances. A *task* is defined by three constants: an ID, an application number, and a configuration assignment. If for two instances all parameters except for the configuration constant are identical, these present a single instance which is common to both systems. Similarly, instances can be found which only exist in one configuration.

For all other component types in the specification, the procedure is the same.

```
% Equal tasks in both configurations
equal_task(task(NUM,A,child)) :- task(NUM,A,child), task(NUM,A,parent).
% Missing tasks in child configuration with regard to parent configuration
missing_task(task(NUM,A,child)) :- not task(NUM,A,child),
    task(NUM,A,parent).
% Added tasks in child configuration with regard to parent configuration
added_task(task(NUM,A,child)) :- task(NUM,A,child),
    not task(NUM,A,parent).
```

Code snippet 1: The analysis of all instances of the component type *task*

4.2 Analysis of the Implementations

Besides the specification, the implementation of the parent configuration is clearly determined and available. To be able to take on a design decision from the previous system, the prospective decisions made to generate a solution candidate for the child configuration have to be evaluated. Like Fig. 2 illustrates, the results from the evaluation alongside with the optimization objectives influence the selection of favorable solutions.

To compare two implementations, each decision, including the allocation, binding, routing and scheduling, is considered. In the following Code snippet 2, the decision on the task binding is taken as a representative case.

```
% Equally decided bindings in both configurations
equal_bind(bind(M,task(T,A,child),processor(R,child))) :-
    bind(M,task(T,A,child),processor(R,child)),
    bind(_,task(T,A,parent),processor(R,parent)).
% Not equally decided binding for equal tasks in both configurations
not_equally_bind(bind(M1,task(T,A,child),processor(R1,child)),
    bind(M2,task(T,A,child),processor(R2,child))) :-
    bind(M1,task(T,A,child),processor(R1,child)),
    bind(M2,task(T,A,parent),processor(R2,parent)), R1!=R2.
% Missing binding in child configuration with regard to parent configuration
missing_bind(bind(M,task(T,A,child),processor(R,child))) :-
    bind(M,task(T,A,parent),processor(R,parent)), missing_task(task(T,A,child)).
% Added binding in child configuration with regard to parent configuration
added_bind(bind(M,task(T,A,child),processor(R,child))) :-
    bind(M,task(T,A,child),processor(R,child)), added_task(task(T,A,child)).
```

Code snippet 2: The analysis of all instances of the decision on the binding

It is identified by a mapping id and a corresponding task mapped to a certain processor, each belonging to a configuration. Analogical to the scheme presented in section 4.1, three result types are expected: *equal*, *missing*, *added* and evaluated from the perspective of the child configuration. The similarity information generated by use of the terms in Code snippet 1 is used to decide on similarities in the implementations. For example, a task which was added or deleted cannot be bound equally and therefore causes an *added* or a *missing* bind. Furthermore, the type *not_equally* is introduced to ensure an unambiguous evaluation. Otherwise, for two configurations, which might have tasks and processors in common, but do not share the same binding decision, bindings might be simultaneously classified as *missing* and *added* and, this way, be counted twice. For all other decision types in the implementation, the analysis is done likewise.

4.3 Use of Heuristics during Design Space Exploration

Through analyzing the specifications and the implementations, an extensive knowledge is built up which is particularly useful for the development of the new derived product. It is assumed that, in the search space, a good solution for the child configuration is to be found near to the design point of the implementation of the parent configuration. Hoping that the optimal solution of the DSE for the child configuration is similar to the implementation of the parent configuration, the gained similarity information is used in heuristics to select appropriate design decisions from the previous system and set them as an initial design point. Thus, the exploration starts in a defined area of the search space and is controlled. At the same time, the search space is not restricted and no solution is excluded. If there are similarities in the specifications, all related decisions made in the development of the previous version are adopted and every variable assignment is preferably decided as previously done for the parent configuration.

Considering the example from section 4.2, the similarity information about the decision on the task binding is taken up in a heuristic in Code snippet 3.

```
1 % Highest priority for deciding the binding equally to the parent configuration
2 #heuristic equal_bind(bind(M,T,R)). [23,true]
```

Code snippet 3: The heuristic influencing the decision on the binding

In the implementation of the heuristic in ASP, a so-called *modifier* is used. It prioritizes the individual term in a way that it, if possible, is assigned a specified value (*true* or *false*) and evaluated earlier during the DSE. In the code sample, the decision to set a binding equally compared to the parent configuration is assigned a static priority of *23*. At the same time, the decision on the allocation is indirectly made when a hardware resource is used in a binding. In case of a task that is only specified in one system, it is impossible to decide the binding identically. Hence, another heuristic is set whose aim is to, at least, bind the task to a common processor. Thus, the allocation of a new and additional resource might become superfluous, if no task is bound to it in the final solution.

The design decision on the binding is considered in the following step of the routing. If there is a common binding of a task on a processor, the communication

path to and from that processor is adopted from the parent configuration. This approach is given in Code snippet 4. The heuristics deciding on the scheduling is considering the execution order of the tasks and similarly implemented.

Further, the synthesis steps are executed in order. According to their priority, the binding decisions are determined first, followed by the routing and the scheduling step. It is conceivable as well to decide on all equal elements first and then to consider the differences. This offers the advantage of a clear separation between the similarities and the differences.

```
1 % If binding was equally decided in both configurations, decide for the same
      routing like in the parent-configuration
2 #heuristic equal_reached(reached(comm(T1,_,_,child), processor(P,child),
      router(R,child))) : equal_bind(bind(_,T1,P)). [13,true]
3 #heuristic equal_reached(reached(comm(_,T2,_,child), router(R,child),
      processor(P,child))) : equal_bind(bind(_,T2,P)). [13,true]
4 #heuristic equal_reached(reached(C,router(R1,child),router(R2,child))). [13,true]
```

Code snippet 4: The heuristic influencing the decision on the routing

Starting with a good implementation for the child configuration consisting of adopted design decisions from the parent configuration, a faster converge to optimal solutions is expected.

5 Experiments

The implementation of this project consists of ASP and C++ code as well as bash scripts for the project execution. The tool clingo is used in version 5.2.2 [13]. To set time stamps and to interrupt a DSE at a certain time (timeout) the tool runsolver in version 3.4 is used [14]. The experiments introduced subsequently are tested on Intel Core i7-4770 CPUs with x86-64 architecture and 32 GiB RAM. The surrounding environment is Ubuntu version 16.04.7 LTS.

5.1 Experimental Setup

As a setup, 24 parent instances of different characteristics are taken from a set of test cases generated by an ASP-based benchmark generator [12]. A 3×3 grid structure, consisting of nine routers bidirectionally connected to each other and additionally to one processor each, is common to all configurations while their application graphs are structured differently. These are generated as "series-parallel graphs" (SPG) having twelve different sizes in the range of 17 to 115 tasks.

To assume good solutions for the parent instances as a basis for the experiments, a DSE for each has been executed for 48 hours. Randomly, one of the best but not necessarily optimal solutions is taken as implementation of the parent configuration. From each parent specification, ten modified child specifications are generated. The modification is composed of a randomly decided combination of different changes including the deletion, addition or exchange of components. In the experiments, either tasks as representatives of the software side

Table 1: Overview of the modification classes specifying the test cases

Change hardware elements			Change software elements			Combined changes		
	p_t	p_p		p_t	p_p		p_t	p_p
I	0	20	V	20	0	IX	20	20
II	0	40	VI	40	0	X	40	40
III	0	60	VII	60	0	XI	60	60
IV	0	80	VIII	80	0	XII	80	80

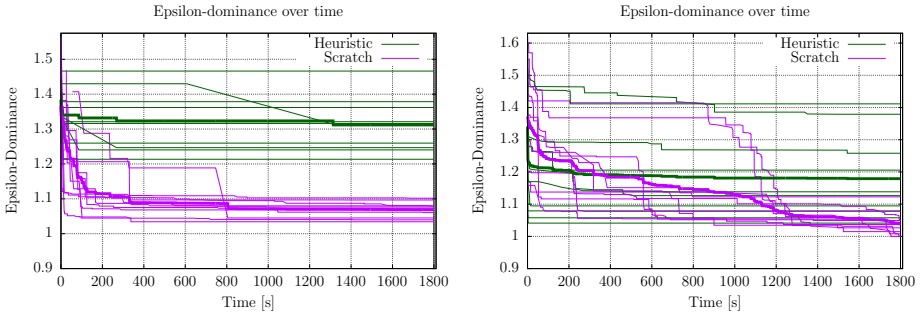
(p_t), processors as elements of the architecture graph (p_p), or combinations of both are considered. Table 1 gives an overview about the chosen test cases.

In total, we have conducted $2 * 24 * 12 * 10 = 5,760$ DSE runs for up to 30 minutes to explore the child configurations. This relatively short time was chosen to be able to consider different modification classes and a sufficient number of randomly generated modifications to obtain a generally valid statement. The resulting fronts are evaluated concerning their ϵ-dominance [16]. Therefore, a reference front, each consisting of the best solution front found up to a timeout of the DSE with and without the use of heuristics, is generated. It is considered as the optimal solution front. Additionally, all intermediate solutions found during the DSE are assigned with a time stamp and evaluated as well. The results together with their corresponding time stamps are plotted to identify the quality improvements over time. Per test case and per parent configuration each, an average curve is presented along with the individual results for the ten children.

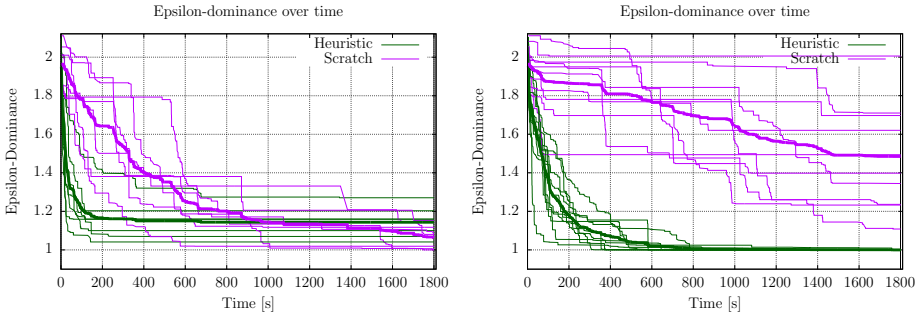
5.2 Experimental Results

The test execution results in 24 diagrams per test case, each illustrating the progression of the ϵ-dominance during the DSE with and without heuristics over time for one parent instance. Until a timeout, which is set as a vertical line at 1,800s, is reached, each curve approximates an ϵ-dominance equals one. Matching this value indicates that the respective solution front is covering the reference front in every design point. Obtaining this result at an early time is the desirable outcome. All in all, four types of curve progressions are identified and pictured in Fig. 3.

The first and fourth type mark a course where either the scratch or the heuristic curve is obviously faster approximating the value one. Whereas for the third type, the heuristic curve is developing to low values fast, but is overtaken by the scratch curve during the exploration. The second type represents a case which does not allow a clear determination. Figure 4 aggregates the occurrences of the four types given in Fig. 3 for all modification classes and considering the 24 configurations. The types 1 (= purple), 2 (= yellow), 3 (= light green), 4 (= dark green) are colored and rated in ascending order with type 4 indicating the superiority of our proposed approach. Having this overview, a considerable trend can be detected. The usefulness of the usage of heuristics depends on



(a) Type 1 - Scratch better than Heuristic (b) Type 2 - Both with similar performance



(c) Type 3 - Heuristic with good start is overtaken by Scratch (d) Type 4 - Heuristic better than Scratch

Fig. 3: The four resulting average curve types

the considered configuration. Noting that an increasing configuration number signifies a larger amount of applications and tasks, the use of heuristics tend to work better for large configurations. With an increasing size of a configuration, the time to exhaustively explore the respective design space grows exponentially. Thus, controlling the DSE by the use of any heuristics is essential to find good solutions in a reasonable time. For large configurations, the proposed heuristics provide excellent results, but it can not be evaluated how close these are towards the real Pareto-optimal solution front because the design space is hardly explored after 30 minutes. At the same time, the DSE for the configurations 1 and 2, mainly finishing within the given time, shows satisfying results as well.

Considering the kind of modification, a few differences for cases with a lot of changes like III, IV, VII or XI are visible, but in overall no clear classification is identifiable.

In a second evaluation, the results for the configuration 11, which contains 55 tasks distributed over two applications, are analyzed in more detail. Table 2 lists for every change type the time it takes to reach a specified ϵ-dominance value. A table entry consists of a number of the explorations reaching the respective level,

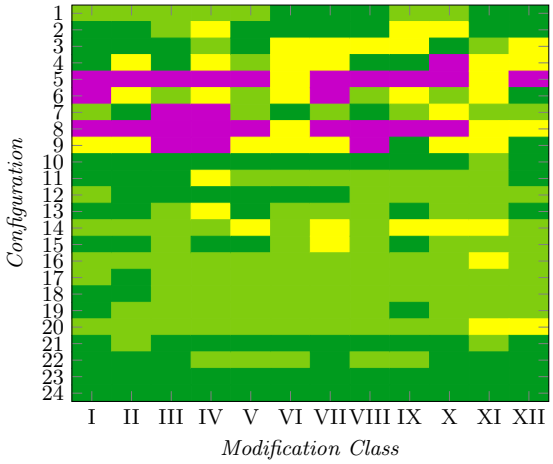


Fig. 4: Summary of the occurrence of the result types from Fig. 3 (Purple - Type 1; Yellow - Type 2; Light Green - Type 3; Dark Green - Type 4)

an average time value and a corresponding standard deviation. The behavior of the DSE with and without heuristics is compared.

Thereby, two aspects become visible. At first, not all explorations even reach an ϵ-dominance equal to two because large changes on the architecture graph (p_p and $p_p + p_t$), mainly in case of deletion of processors, cause the creation of unsatisfiable child instances. Furthermore, the heuristic DSE for satisfiable child instances, which are derived by purely changing the hardware side (p_p), provides significantly better results. This becomes more clear, the closer an ϵ-dominance equal to one is approximated. More heuristic instances are reaching lower stages and the results are found up to 1,000 times faster with lower deviations. These cases are perfect examples for the type 4 from Fig. 3.

Secondly, the results when considering only the modifications on the application graph (p_t) are ideal representatives for type 3 from Fig. 3. By using the own heuristics, the first three levels are reached within a few seconds. But as time goes on, the DSE from scratch is more successful. At the same time, most cases of both exploration types are finally not reaching an ϵ-dominance of one, which means that different design points exclusively were found and several valid implementations with good properties exist.

In general, Table 2 shows that the average times and the standard deviations for the first solutions from the DSE with and without heuristics are similar. If the heuristic DSE reaches a low ϵ-dominance, it takes a significantly shorter time. Likewise, the deviation is lower. This is an important result, showing that good results can be found at an early exploration stage without the necessity to search the entire design space.

Table 2: Comparison of the influence of different change types

		2		**1.5**		**1.3**		**1.1**		**1.0**	
		$\mathcal{S}$	$\mathcal{H}$	$\mathcal{S}$	$\mathcal{H}$	$\mathcal{S}$	$\mathcal{H}$	$\mathcal{S}$	$\mathcal{H}$	$\mathcal{S}$	$\mathcal{H}$
p_p	20	10	10	10	10	10	10	5	9	0	4
		0.70 s	0.73 s	0.70 s	0.73 s	101.62 s	0.81 s	566.03 s	0.91 s	-	315.77 s
		6.26 %	7.59 %	6.00 %	6.59 %	182.06 %	8.1 %	120.87 %	7.71 %	-	87.74 %
	40	9	9	9	9	9	9	6	8	0	1
		0.66 s	0.69 s	3.84 s	0.70 s	70.11 s	0.76 s	822.49 s	0.87 s	-	558.36 s
		7.75 %	7.53 %	248.05 %	8.15 %	91.26 %	6.78 %	63.46 %	14.90 %	-	-
	60	10	10	10	10	100	10	6	10	0	5
		0.66 s	0.67 s	0.66 s	0.67 s	101.28 s	0.75 s	653.43 s	0.96 s	-	803.26 s
		11.56 %	14.14 %	11.56 %	14.14 %	138.32 %	23.35 %	94,56 %	22.85 %	-	81.10 %
	80	5	5	5	5	5	4	1	3	0	0
		0.64 s	0.63 s	0.80 s	0.63 s	134.22 s	0.66 s	849.31 s	0.76 s	-	-
		12.66 %	13.28 %	46.39 %	13.28 %	84.79 %	11.46 %	-	15.63 %	-	-
p_t	20	10	10	10	10	10	10	9	6	1	0
		0.74 s	0.76 s	6.85 s	0.80 s	175.78 s	0.93 s	694.56 s	14.43 s	1573.48 s	-
		12.97 %	7.89 %	132.70 %	5.77 %	105.01 %	10.76 %	78.29 %	227.26 %	-	-
	40	10	10	10	10	10	9	8	7	5	1
		120.73 s	0.93 s	129.55 s	0.94 s	185.42 s	1.05 s	752.76 s	1.64 s	1627.63 s	49.54 s
		146.54 %	23.44 %	132.63 %	22.50 %	124.74 %	19.56 %	71.24 %	47.87 %	14.51 %	-
	60	10	10	10	10	10	10	9	8	4	0
		0.77 s	0.81 s	52.94 s	0.90 s	175.09 s	1.74 s	453.64 s	305.82 s	1377.95 s	-
		20.52 %	18.08 %	172.62 %	16.33 %	111.62 %	112.44 %	53.79 %	174.50 %	25.85 %	-
	80	10	10	10	10	10	10	8	6	1	2
		0.81 s	0.81 s	36.85 s	1.37 s	215.07 s	4.25 s	591.55 s	6.68 s	1380.83 s	6.50 s
		31.61 %	22.40 %	10.14 %	65.70 %	132.61 %	177.05 %	85.23 %	154.22 %	-	54.11 %
$p_p + p_t$	20	10	10	10	10	10	10	9	9	1	1
		0.80 s	0.73 s	20.09 s	0.74 s	123.77 s	0.89 s	328.02 s	1.13 s	1739.00 s	407.36 s
		36.66 %	11.02 %	154.13 %	10.09 %	144.38 %	19.69 %	72.60 %	33.53 %	-	-
	40	10	10	10	9	10	9	5	5	2	1
		0.80 s	0.74 s	9.68 s	0.76 s	116.54 s	0.89 s	372.73 s	1.13 s	1730.17 s	1042.53 s
		32.34 %	13.67 %	108.62 %	17.35 %	78.60 %	21.32 %	120.13 %	23.52 %	0.84 %	-
	60	5	5	5	5	5	5	3	1	1	0
		120.51 s	184,54 s	120.52 s	184.57 s	162.42 s	186.84 s	590.29 s	1.21 s	1099.08 s	-
		112.08 %	217.65 %	112.07 %	217.60 %	82.09 %	214.29 %	42.76 %	-	-	-
	80	6	6	6	6	6	5	3	3	1	0
		152.49 s	0.80 s	188.32 s	0.94 s	412.54 s	1.13 s	435.72 s	5.03 s	1415.28 s	-
		243.92 %	16.71 %	211.23 %	24.13 %	144.05 %	19.42 %	63.92 %	87.86 %	-	-

6 Conclusion

Within this paper, a systematic approach based on ASP to enhance the DSE of embedded systems is proposed. It aims at supporting an evolutionary product design process in the context of *Product Generation Engineering*. Exploiting the similarities between a base system and its derivatives allows to identify parts that can be reused unchanged. The gained domain knowledge is utilized in heuristics to steer the search towards regions in the design space potentially containing solutions with optimal properties.

To ensure a meaningful evaluation of the impact of the used heuristics, an extensive amount of test cases, consisting of a variety of different configurations and several systematically derived child instances, was used in the experiments. As expected, the usage of heuristics in the DSE helps to find good solutions earlier. While small systems are less likely to profit from the introduction of the proposed heuristics, particularly in large system configurations, the application

of heuristics shows a significantly high exploration quality. For the product development, it is not required to find the optimal implementation because that goes along with an inestimable long exploration time and high costs. Much more preferably is a good solution found at an early time. Likewise, in the majority of the test cases, excellent results were achieved just in a few seconds.

Finally, the implementation at hand can be extended by new heuristics and further use cases. The experiments have shown that there is more potential for identifying reusable parts, especially when analyzing how the structure of a configuration is influencing its reusability.

References

1. Albers, A., et al.: Product generation development-importance and challenges from a design research perspective. In: Proc. of ME. pp. 16–21 (May 2015)
2. Andres, B., et al.: Symbolic System Synthesis Using Answer Set Programming. In: Proc. of LPNMR. pp. 79–91 (2013). https://doi.org/10.1007/978-3-642-40564-8_9
3. Duffy, S., et al.: A design reuse model. In: Proc. of ICED. pp. 490–495 (Aug 1995)
4. Fahimipirehgalin, M., Fischer, J., Bougouffa, S., Vogel-Heuser, B.: Similarity analysis of control software using graph mining. In: INDIN. vol. 1, pp. 508–515 (2019). https://doi.org/10.1109/INDIN41052.2019.8972335
5. Gebser, M., et al.: Theory Solving Made Easy with Clingo 5. In: Proc. of ICLP. pp. 2:1–2:15 (2016). https://doi.org/10.4230/OASIcs.ICLP.2016.2
6. Girczyc, E., Carlson, S.: Increasing design quality and engineering productivity through design reuse. In: Proc. of DATE. pp. 48–53 (1993). https://doi.org/10.1145/157485.164565
7. Goens, A., Siccha, S., Castrillon, J.: Symmetry in software synthesis. ACM TACO **14**(2) (Jul 2017). https://doi.org/10.1145/3095747
8. Liang, J.H., et al.: Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers. In: Hardware and Software: Verification and Testing. pp. 225–241 (2015). https://doi.org/10.1007/978-3-319-26287-1_14
9. Lifschitz, V.: What is Answer Set Programming? In: Proc. of AAAI. pp. 1594–1597 (Jul 2008)
10. Neubauer, K., et al.: Supporting Composition in Symbolic System Synthesis. In: Proc. of SAMOS. pp. 132–139 (Jul 2016). https://doi.org/10.1109/SAMOS.2016.7818340
11. Neubauer, K., et al.: Exact Multi-Objective Design Space Exploration using ASPmT. In: Proc. of DATE. pp. 257–260 (Mar 2018). https://doi.org/10.23919/DATE.2018.8342014
12. Neubauer, K., et al.: Exact Design Space Exploration Based on Consistent Approximations. Electronics **9**(7), 1057 (Jun 2020). https://doi.org/10.3390/electronics9071057
13. Potassco: Clingo homepage, https://potassco.org/clingo/, accessed 13 Mar 2021
14. Roussel, O.: Controlling a solver execution: the runsolver tool. JSAT **7**, 139–144 (Nov 2011). https://doi.org/10.3233/SAT190083
15. Simpson, T.W.: Product platform design and customization: Status and promise. AI EDAM **18**(1), 3–20 (2004). https://doi.org/10.1017/S0890060404040028
16. Zitzler, E., et al.: Performance assessment of multiobjective optimizers: an analysis and review. IEEE TEVC **7**(2), 117–132 (2003). https://doi.org/10.1109/TEVC.2003.810758