Optimization of Systems With Nested Design Space

Armin Zimmermann, Ralph Maschotta, and Alexander Wichmann
Systems and Software Engineering Group
Technische Universität Ilmenau, Germany
Emails: see http://sse.tu-ilmenau.de

Robert Hilbrich
German Aerospace Center (DLR)
Berlin, Germany
robert.hilbrich@dlr.de

***Abstract*—Finding a good solution to an engineering problem under given constraints can be formulated as an optimization problem over a set of parameters. Existing (semi-)automatic methodologies for this task usually assume that these design parameters form a homogeneous set. The paper analyses the structure of design space and introduces the formal concept of *existence-dependent parameters*, which appear for instance in the optimization of system architectures. The corresponding *nested optimization problems* can be solved more efficiently considering the dependency relations, and will often require a mix of models and optimization algorithms. The paper proposes a completely model-driven integration methodology for such problems and a prototype tool combining several existing domain-specific tools. An application example from the area of distributed embedded systems design of hardware architecture and software mapping demonstrates the applicability.**

I. Introduction

The engineering of complex systems is still challenging and costly. Despite our technical and methodical advancements, the construction of a complex system with intricate requirements still bears risk and uncertainties with regard to its outcome. In later engineering phases, such as testing and validation, the automation of tasks has already significantly increased productivity and quality. Unfortunately, tasks in early engineering phases, such as the system design, cannot be automated in a similar fashion. They often require creativity, experience-based solution adaption, and the making of complex decisions for which it is not clear how to derive a perfect answer. Therefore, these "creative tasks" are still predominantly conducted by human engineers. Of course, they benefit significantly from best practices, suggested development phases and sub tasks, as well as rules on how to structure problems and tackle open questions. Domains in which systems design is considered a key factor include automotive, avionics and manufacturing. Systems in these areas are engineered to implement a complex interplay between mechanical elements, electronic components, as well as (embedded) software. Therefore, their design is especially challenging and cannot be easily formalized due to its complexity and interdisciplinary nature. If we abstract from the individual field of application, one of the main remaining certainties are established design process steps as a means to systematically tackle engineering problems in general. This is thus one of the main roads followed in comprehensive books on systems design [1], [2], [3], [4].

Among the main phases and tasks of systems design are requirements elicitation and analysis, finding viable engineering solutions to satisfy requirements, and finally to select the best design among the possible ones based on functional and non-functional properties of the system as well as optimization goals. Non-functional properties of a design (i.e., reliability, energy consumption, size and weight, ...) are usually hard to predict in early design phases as they are *emerging* as a result of the integration of the software and hardware architecture. Therefore, they cannot be analyzed by examining a single component in isolation. Instead they can only be assessed properly after the system has been built and integrated. At that time, a redesign of the system to eliminate the design flaws is often very expensive or even too late. Model-based systems engineering (MBSE, [5]) and corresponding tools [6] support the human engineer by allowing him to predict non-functional properties based on mathematical or simulation models, thus reducing the risks in early design decisions significantly. For safety-critical systems and highly reliable designs such properties are of equal importance as the functional requirements, especially when a certification is necessary before use – as it is often the case in train control or avionic systems.

The *design space* of a system (c.f. Section II) denotes a geometrical understanding of the possible design variants in a multi-dimensional space. Algorithms for design space exploration and optimization treat this space usually as a homogeneous space, in which restrictions and dependencies lead to forbidden regions. In most cases, they can be formally described by certain constraints.

However, we argue in this paper that the design space for complex systems (such as safety-critical embedded systems) is rarely homogeneous. Instead, it is often build as a hierarchy of interdependent design parameters — a *nested design space*. We show that such a nested design space can be explored efficiently with model-based optimization techniques despite its complexity. This type of problem occurs frequently in the optimization of system architectures [7], [8]. An important example are multi-processor embedded systems (c.f. [9]), for which both an optimal hardware architecture as well as a fitting software allocation and mapping has to be found in many application areas. We propose a method that treats these two subordinated problems separately in their individual domain-specific tools, but introduce a work flow and prototype tool integrating their optimization in an elegant way by reusing the (meta-)model information of the specific domains.

The paper is structured as follows: An overview of related work is given subsequently. Section II covers background on

design space (optimization) and introduces some terminology and notation. Section III assesses types of design parameters as well as the corresponding design space structures, and introduces a formal treatment of existence-dependent design parameters. The special subtype of nested problems is identified, for which a workflow is proposed and a prototype software tool introduced in the subsequent Section IV. An application example demonstrating the benefits of this approach is given in Section V, before some conclusions are provided at the end.

The contribution of the paper includes a novel characterization of design space types and structures, which allows a systematic mapping of corresponding optimization methods. Moreover, we introduce a design space optimization method that allows to integrate models and software tools for nested design space problems in a completely model-based fashion without the need of significant programming efforts.

Related Work

There is a vast amount of literature on design-space exploration, of which only a few can be covered here. An overview of multi-objective optimization tools is given in [10]. Model checking can be used as a method to explore feasible parts of the design space, while improving the efficiency by reusing variable orderings [11]. One practical way to reduce design space size is to introduce user constraints and approximations on variables; however, this may lead to issues covered in [12]. Efficient methods for design space exploration of embedded systems have been covered in [13].

An important application area of mapping (software functions to hardware resources) problems are automotive systems. One proposed method in the literature represents system models in SysML, and applies design space exploration methods to synthesize deployments from logical (platform-independent) system models to technical (platform-specific) system models [14]. A specialized framework for optimal computer architecture selection and mapping has been presented in [15], but it is restricted to the special case of deep neural networks. A tool for scenario-based exploration of embedded MPSoC (multi-processor systems on chip) mapping was developed in the ARTEMIS project [16]. Similar approaches for MPSoC designs are described in [17], [9], [18], [19], [20]. The problem of traversing the design space of mapping tasks to electronic control units in combination with feasibility checks and optimization goals is addressed in [21], [22].

II. Background on Design Space Optimization

Numerous decisions have to be taken during the design of a system, each characterized by selecting one out of (possibly many) variants of a system aspect. The set of variants S_i of each of these i decisions ($i = 1, \ldots, n$) can be understood like a domain or range of values of every decision variable p_i, i.e. $\forall i : p_i \in S_i$. Such variables p_i are usually denoted as *design parameters*, which may be independent in the simplest case, and we denote the set of all of them for a certain system design process as P (although this set may not be known before the top-down design refinement has finished). For real-world systems, it is already a challenging and elaborate task to derive not only major decision variables, but also a significant set of alternative values for each variable. Each individual system design variant $\mathbf{p}$ is then a vector of values $(p_1, \ldots, p_n)$ for all design parameters, which can alternatively be understood as a function assigning a value to each design parameter:

$$\forall p_i \in P : \quad \mathbf{p} : p_i \rightarrow S_i \tag{1}$$

If we imagine each of these n parameters as one dimension of a geometrical space, the corresponding set of all (theoretically) possible design solutions $\mathbf{p}$ is the n-dimensional *design space* of all combinations of these variable values $DS = S_1 \times S_2 \times \ldots \times S_n$. The main problem of finding a good design is then its size (without restrictions) of $|DS| = \prod_{i=1}^{n} |S_i|$.

Obviously not every parameter combination constitutes a *feasible* design variant. A design variant is only feasible if it fulfills all mandatory functional and non-functional requirements as well as system-level constraints (e.g. from the laws of physics or due to standards, certification or laws). This can be formalized as a Boolean function $CF : DS \rightarrow \{\mathrm{True}, \mathrm{False}\}$ returning for each possible parameter combination if it is feasible or not, resulting in the subset

$$DS^{feasible} = \{\mathbf{p} \in DS \mid CF(\mathbf{p})\}$$

Assessing the feasibility of a parameter combination is often a significant challenge in its own right, which sometimes even requires the development of prototype systems.

Within this context, *design space exploration (DSE)* denotes the task of generating and evaluating a set of feasible design variants (parameter combinations) of the system to be built. However, systems engineers are interested in finding the best design variant out of the remaining feasible ones. The decision about which remaining design is the best has thus to be based on a weighted evaluation of the benefits of (non-mandatory) functional and (usually quantitative) non-functional properties. There is a huge variety of models, evaluation methods and tools available for this task in the broader field of performance evaluation (c.f. e.g. [23]).

The general task of a systems designer can thus be understood as a combined *design space exploration* and *design space optimization* problem: given the requirements as constraints, find the best solution under a certain profit function f_{profit} over the feasible part of the design space returning a real value to be maximized:

$$f_{\mathrm{profit}} : DS^{feasible} \rightarrow \mathbb{R}$$

This function requires an understanding of the mathematical relation from the design parameters to the values of quantitative properties of the system. The results can then be assessed for their benefit with a function mapping real values into unit-less comparable numbers for use in a weighted sum that captures the trade-offs typical for all designs for instance with a cost/benefit analysis [4], [13] (often termed multi-objective optimization). In fact, practical design optimization without trade-offs is hard to imagine and would result in obvious

designs that are on the boundaries of design parameters. Such cases may arise from ignoring certain stakeholders or can be a sign of forgotten non-functional aspects of a system design.

The search for the parameter set

$$\mathbf{p}^{\mathrm{opt}} = (p_1^{\mathrm{opt}}, p_2^{\mathrm{opt}}, \ldots, p_n^{\mathrm{opt}})$$

maximizing the profit function

$$\mathbf{p}^{\mathrm{opt}} = \arg \max_{p \in DS; CF(p)} f_{\mathrm{profit}}(p)$$

is a classical optimization setting. A direct optimization (algorithm) is only applicable if the function returning the value(s) of non-functional properties can be symbolically described with a simple function. For instance, if the system, its constraints and benefit can be described by linear expressions, the problem may be described as a simple linear programming problem (LPP) for which efficient algorithms exist. However, in the case of complex and especially dynamic systems, a model and evaluation method is usually needed to capture the intrinsic dependencies. This only allows to evaluate the values of quantitative properties one-by-one for individual design parameter settings, which is obviously very time-consuming because of the computation time per evaluation and the sheer number of possible variants (i.e., the size of the design space). This well-known problem has led to the development of *indirect optimization* methods based on heuristics such as simulated annealing, genetic algorithms, and taboo search to speed up the search for a (near-)optimal solution. Sometimes the goal is not to derive the provably best design variant, and a subset of best/near-best configurations should be found (DSE subsetting).

III. A Characterization of Design Space Structures and Their Impact on Optimization

Theoretically possible values of a design parameter have to be synthesized by applying existing solution patterns, varying the type or number of system elements, or by selecting a certain technical solution for a required function. This is the essence of the "creative tasks" conducted by human engineers during the system design phase. As a result, the value range S_i of a parameter p_i can be characterized by the following decision types [8]:

1) A typically real-valued interval of a non-functional configuration parameter specifying a property such as speed, length, size, bandwidth, reliability etc. for an element of the system design;
2) An (integer) number of elements or resources of a certain type in a system, often restricted to an interval of values;
3) An optional item or decision with two outcomes (this may be viewed as a special case of integer with values 0 or 1);
4) A list of possible choices for an element, architecture, or other property (similar to an enumeration type);
5) Or (as a special case) a derived parameter that is variable in its local setting, but depends uniquely on other decisions.

In real-life scenarios, many of the above-mentioned design decisions resulting in parameter value p_i choices influence each other, such that the variables cannot be chosen independently. One of the main reasons is closely connected to the top-down design approach: System- and subsystem-level requirements address different abstraction layers of a system leading to a hierarchy of design decisions. A choice made on higher levels typically results in additional requirements on lower levels. Therefore, picking a value for a decision variable p_i on a higher level restricts the choice of values for a decision variable p_j on a lower level of design. Such a dependency may be rooted in the joint contribution to a certain property (for instance security or energy consumption).

However, a very different type of dependency stems from system designs where the structural architecture can be varied, and local design parameters emerge or vanish because of such choices. An example is the design of a distributed system, in which the communication network protocol and structure can be chosen freely. Let's assume that the designer may choose between a wireless solution and a (wired) bus architecture. Reliability concerns may then require to choose between several possible methods to ensure reliable packet transfer in the radio-based case, which may not be necessary because of the sufficient quality of a wired network. In such examples not only the *values* of design parameters depend on each other, but even the *structure of the design parameter set* itself. In this case, the function $\mathbf{p}$ describing a system design given in Equation 1 will only be *partial*, i.e., defined just on a subset of design parameters P. The set of design parameters necessary or relevant for a certain system design is thus variable and not a constant full set P in such cases. Describing such problem types requires models that go well beyond a vector of numerical values of design parameters [24], [25], [7], [8].

The corresponding type of relation between two design parameters is formalized as follows:

Definition 1: A design parameter p_k is said to be **existence dependent** on another parameter p_i of the same system if it is only relevant or meaningful if the latter has a certain value out of (or range) $S_i' \subset S_i$. We can also say that a value for p_k has to be chosen in every design variant in which $p_i \in S_i' \subset S_i$, otherwise p_k will be irrelevant as it usually specifies parameters of system components that are not part of this variant. This relation will be denoted by $p_i \succ p_k$ (p_i *dominates* p_k or p_k is existence dependent on p_i, $p_i \prec p_k$). The terminology draws on a similar dependency in the design of relational data bases, in which existence dependency describes entities that have to exist if another entity does; our definition is thus stricter as it is also based on values.

No design parameter dominates itself, thus the $\succ$-relation is irreflexive. It is also asymmetric as no two design variables can mutually dominate each other. Moreover, it is transitive because obviously if $p_i \succ p_j$ and $p_j \succ p_k$ then also $p_i \succ p_k$. The $\succ$-relation thus forms a *strict partial order* on the set of design parameters.

When reasoning about similar sets of such design parameters, two parameters belong to a joint subset if their existence

dependency relation to all other parameters is equal[1]:

$$p_i \simeq p_k \longleftrightarrow \left(\succ^+(p_i) = \succ^+(p_k) \wedge \prec^+(p_i) = \prec^+(p_k) \right)$$

The $\simeq$-relation on design parameters is obviously reflexive, symmetric, and transitive, and thus an equivalence relation on P. If we take the resulting equivalence classes $S^1 \ldots S^m$ with $\bigcup S^i = S$ as nodes of a graph DG and add edges according to the $\succ$-relation of class representatives, the result is a *directed acyclic graph* (dag) DG between subsets of design parameters that are not dominating each other (i.e., usually belonging to the same level and component of the system)[2]. The graph follows the top-down hierarchy of subsystems and corresponding structural design decisions during refinement, and has a unique root node S^1 that is the starting point of systems design at the highest level.

We term a resulting design space with variable structure (i.e., $\left|\left\{S^1 \ldots S^m\right\}\right| > 1$) as being **nested**, as reasoning about settings for existence dependent parameters is meaningful only under a given parameter choice of the dominating ones. System design problems without this issue may be seen as *flat*, as there is no obvious way of knowing which design parameters dominate others.

What can we gain from this for design space optimization? If the engineer was able to determine the hierarchy among the p_i by carefully analyzing the $\succ$ relations or simply from following the iterative top-down design refinement, the exploration of the design space can omit other parameters which are not applicable for a certain component design. Of course, there is a strong need for improved tool support to facilitate the understanding of the design choices and their implications.

An overview of design space exploration methods is, for instance, given in [13]. Obvious (and usually impractical) approaches include manual exploration and exhaustive search. Direct numerical optimization is possible for very restricted problems only. It does not work if the benefit function has jumps, for instance, in which case they may be described with an integer LPP in simple cases, for which the solution is already NP-complete. Indirect optimization requires heuristics that can be characterized as *black box randomized approaches* [13]. They work in a generate-and-test manner by randomly choosing a proposed design variant and assessing its feasibility and performance (benefit) afterwards.

However, such methods suffer largely from nested design spaces: Theoretically it may still be possible to describe the design space with a cross product of the values of all variables as $\mathbf{p} \in S_1 \times S_2 \times \ldots \times S_n$, but this ignores their dependencies. A standard design space exploration method would thus be hampered by traversing large parts of DS, wasting time on numerous unnecessary or obviously impossible solutions $\mathbf{p}$. Moreover, the usual assumption is a neighborhood relation between "similar" solutions. This can be approximated by the length of an assumed n-dimensional vector between two solutions in the design space, which does not apply well to nested problems because of their variable structure.

We propose to apply optimization methods individually for each nested subspace, and to embed the optimization algorithms themselves in their loop structures. This approach can be directly controlled by the dependency graph DG that implies a weak ordering on design parameters. The root parameter class having no successor in the dag DG is explored in the outermost optimization loop (which would be the only one in a flat problem setting). For each selection of the set of parameters in the current subset corresponding to an element of DG, the directly dependent parameters have to be iterated next with their own heuristic. In fact there may be problem structures in which a lower-level design decision can be solved directly, although it is embedded in a heuristic search at a higher level. This type of branch-and-bound strategy will always be possible, following the weak ordering of the design parameter classes captured in DG.

This allows also to treat different system description levels individually with models from their corresponding abstraction level, which follow the hierarchy of design decisions. Such a hybrid method will require an integrated framework that may apply different tools to certain parts of the optimization problem. A typical example is the automated hardware architecture optimization of embedded systems, into which the allocation and mapping of functions (software components) is embedded (nested) [2], [1], [27]. We propose a tool framework in the subsequent section that allows the integrated treatment of nested design spaces and is generic in its applicability by using a model-driven UML approach. In our example, a structural optimization of hardware architecture is done with a standard indirect heuristic, while the embedded problem of mapping and allocation is tackled efficiently with a constraint solver that generates feasible solutions of its nested sub problems.

IV. A Prototype Tool for Nested Design Space Optimization

This section describes how the proposed method is implemented as an integration of our previous work on architecture optimization [25], [8] and software mapping [28]. Figure 1 depicts the overall optimization loop with a nested call to a constraint solver for the mapping problem. The approach of model-driven optimization is used to find an optimal architecture for a considered system. A heuristic such as simulated annealing etc. is applied in order to select a valid architecture variant. This variant is transformed into an evaluable model, which is passed though external evaluation tool. The resulting evaluation value serves as a rating of the variant's benefit f_{profit}. The heuristic uses this value to generate new parameter sets $\mathbf{p}$ to converge to the optimum $\mathbf{p}^{\text{opt}}$. In this paper we restrict ourselves to a problem in which the inner mapping solver returns the best solution for a given architecture, and the overall optimization is done on parameters which can be derived without a detailed feasibility check or simulation.

[1] with $\succ^+(.)$ denoting the transitive closure of the existence dependency relation over P for a given design variable and $\prec^+(.)$ vice versa

[2] A different dependency graph for design parameters is constructed in [26] to speed up the numerical evaluation of Pareto-optimal system on chip architectures.

We present a combination of an automated indirect optimization of system architectures based on the modeling and solving of mapping problems by using the Architecture Synthesis for Safety-Critical Systems (ASSIST) Tool Suite [29]. ASSIST allows a systems engineer to automatically construct and optimize mappings between software components and the hardware resources. Its focus is on the design of complex control systems in the avionics domain.

The textual specifications in ASSIST conform to a domain-specific language (DSL) which is defined using Eclipse XCore. The ASSIST DSL allows specifying the structure of a system model based on elements like *Boards*, *Processors* or *Applications*. It is used to define a mapping problem between the capabilities and resources of the hardware architecture and the resource demands of the software applications. It also allows the user to specify constraints, such as co-locality and dislocality, in order to address reliability or performance requirements. ASSIST uses the constraint solver Choco [30] internally to find feasible (and optimal if configured accordingly) solutions.

Complex heterogeneous systems are modeled using different kinds of domain-specific languages for different parts of systems. Hence, various models and tools are used to represent and analyze different system aspects. If the tools use standardized meta meta-models like ecore or MOF/UML, a collective meta-model for the different domain-specific parts of the system can be created, which enables the seamless integration and interaction between different tools for different parts of the system. This allows the analysis of different aspects of the system using different tools based on the same model, and becomes possible because of available standard transformations between standard meta meta-models.

The system design is modeled using standard UML first, in particular with class diagrams. For the automatic optimization of this system, a description of valid variants is required. The system model is enriched accordingly with variant-specific stereotypes defined inside a UML profile called Variant Profile [8]. Each stereotype indicates a variant type and can be configured in detail using stereotype properties. The resulting system architecture variant model includes all conceivable alternatives of the system.

The optimization heuristic generates explicit solutions of a design problem. Each solution is then evaluated for feasibility by the ASSIST tool suite and its embedded constraint solver Choco by trying to construct a valid mapping between software components and hardware resources.

In order to achieve a model-based and streamlined approach, we used the ASSIST meta model as system model after an automatic transformation into a UML-conform model. The system optimization process itself is completely specified in UML. However, the ASSIST meta model is defined with XCore. Using the EMP (Eclipse Modeling Project), both meta-models are based on the same ecore meta meta-model. Thus, it is possible to convert the XCore specifications into an UML specification using standard EMP tools. The complexity of the UML model is higher than the XCore model. Hence, this transformation is possible without losing any information.

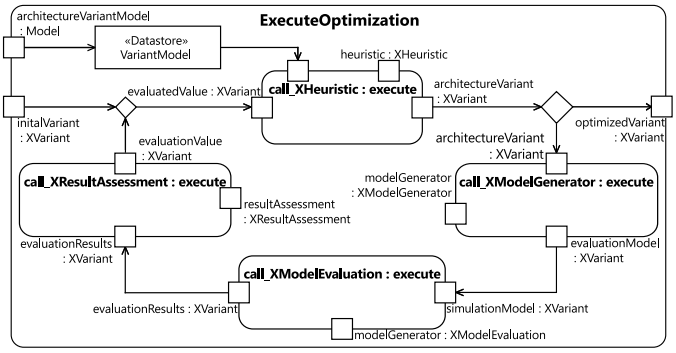

Fig. 1. Execution model of a domain-specific indirect optimization loop

Figure 3 presents the transformed ASSIST XCore classes which are used to build an Architecture Variant Model. The class diagram depicts the relevant classes to build a system model and to specify a mapping problem for the ASSIST tool suite.

The transformed model conforms to the UML and thus the Variant Profile can be applied, such that architecture variants of this system can be specified. The resulting architecture variant model is the major input for the architecture optimization. Figure 1 represents the top-level behavior of the optimization. It starts with the execution of a simulated annealing heuristic. The UML the meta class *InstanceSpecification* describes instances of a modeled system, which is used to specify the architecture variant.

Each architecture variant has to be evaluated and compared with previous variants. This should be done by executing the ASSIST constraint solver, but this solver cannot be executed with an UML *InstanceSpecification* specification as input. For that, a transformation into an evaluable ASSIST conform architecture model is required.

Figure 2 represents the class diagram for the architecture optimization. The class *Optimization* includes the operation *executeOptimization*, which in specified with the activity diagram shown in Figure 1. Additionally, the class *Optimization* has references to four interfaces defining the components of the optimization. Each interface defines a specific *execute* operation, which is called by the top-level behavior of the architecture optimization. This is possible by using the UML element *CallOperationAction*. As the name suggests, *CallOperationAction* defines that a specified *Operation* of an *Interface* or *Class* should be executed. The fourth interface *XVariant* is used for data exchange.

The action *call_XModelGenerator* in Figure 1 should call an operation defined in the interface *XModelGenerator*. The actual behavior, which is executed by this action, has to be defined inside a class realizing the interface. This allows an easy exchange of the desired behavior. The right middle side of Figure 2 shows that a class *AssistModelGenerator* is used here to transform the *InstanceSpecification*-based architecture variant into the required ASSIST domain-specific language.

The resulting model can be evaluated for feasibility by the

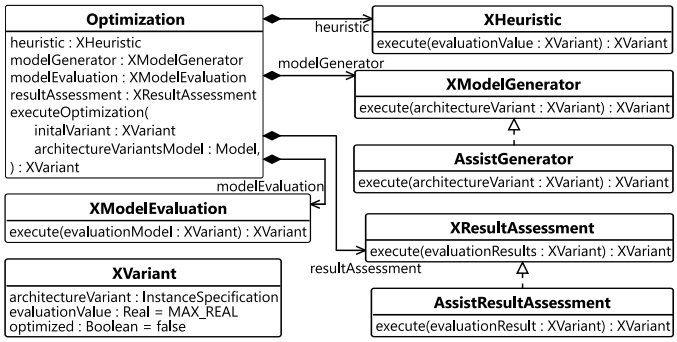

Fig. 2. Optimization classes and ASSIST specialized classes

ASSIST Tool Suite [29]. An already existing model for the action *call_XModelEvalution* can be applied, which starts an external tool and waits until its computation is finished. The *ASSIST Tool Suite* reads the architecture model, executes the internal constraint solver Choco and returns the results of the mapping. Additionally, the results also contain evaluation values which are calculated for valid mappings.

In the next step, the evaluation results have to be assessed in terms of the benefit function f_{profit}. For that, the class *AssistResultReader* implements the interface *XResultAssessment* (right bottom in Figure 2). Two results are possible: first, a valid mapping could not be created for the given architecture variant, in which case it is marked as invalid. The heuristic is executed again and another variant is selected. Secondly, at least one valid mapping exists for the input architecture. The mapping with the best evaluation value is chosen and the corresponding architecture variant will be completed with the mapping information. As long as the termination condition is not satisfied, new architecture variants will be derived from the current best or last one and evaluated afterwards. The best variant is returned as result of the optimization finally.

V. An Example System

The design of a flight management system (FMS) for an airplane is used as an example to demonstrate the intended workflow. It is based on the examples presented in [31], [32]. An FMS supports the pilot by automating a variety of critical in-flight tasks (such as trajectory prediction) and providing accurate information relevant for navigation and flight planning. It has to be able to realize several computationally intensive tasks, while also being highly reliable as it is an essential system for the safe operation of an airplane. Larger flight management systems are often built based on powerful and fault-tolerant processing platforms, such as the Integrated Modular Avionics (IMA) platform [33]. At the same time, the design of a flight management system should also minimize its space, weight and power requirements to increase fuel efficiency. Exploring the design space of a flight management is very challenging for a human engineer due to the large amount of design decisions which have to be made. This level of complexity not only hampers the basic feasibility determination and reliability assessment of a design variant, but also improves the difficulty of optimizing non-functional properties.

A. Architecture Design Space

This application example is used to build a specific architecture variant model, to determine which hardware configuration is optimal with respect to hardware utilization, and by deciding which software applications should be run on which node. Figure 3 represents the developed architecture variant model.

The *AssistModel* includes software applications as well as hardware resources. The software architecture is represented by classes *Application* and *Tasks*. These classes are part of the variant description insofar, that three applications are explicitly specified using a dependency with applied stereotype *countFixedInstanceVariant*. This stereotype is configured such that three instances of class *Application* should be assigned to the property *applications* of class *AssistModel*. Furthermore, these three instances are predefined by object identifiers *application1*, *application2* and *application3*.

These application instances are described using UML object diagram shown in Figure 4. One application is used for a primary flight display. An application for air data is redundantly represented by the other two application instances, which should not be executed on the same processor to improve redundancy.

The hardware architecture is defined using variant stereotype *countFixedInstanceVariant*. Predefined instance specification are not defined for both classes. This is not necessary, because there are no attributes that need to be defined beforehand. The class *Compartment* has only a reference to instances of class *Box*, which has no fixed attributes except the back reference to its owner.

However, *Box* has a reference to class *Board*. Several *Board* instances are predefined and presented in Figure 5. The *board1:Board* instance in Figure 5a represents a "high end" board having many resources. Furthermore, six possible mainframe *Board* instances (*board2:Board* to *board7:Board*) with significantly less resources are defined. Figure 5b presents two of six *Board* instances. The other *Board* instances are identical with regard to their technical properties.

The task of the architecture optimization is the selection of suitable type and number of the boards such that all applications can be executed on fitting resources while minimizing the amount of unused resources. The mapping of the *Application* instances to the selected hardware resources is done by the *ASSIST Tool Suite*. The cost function to be minimized is calculated inside the *ASSIST solver*. The cost of a system variant comprises the scaled maximum free capacity of all *Core* elements plus scaled penalties for unused *Core*, *Processor*, and *Boards* elements.

The application example has been optimized with the prototype tool chain proposed in Section IV. Figure 6 shows temporary results of the cost function as computed by ASSIST during the subsequent optimization loop iterations, each for a different architecture variant as generated by the outer loop and with the optimal task mapping as found by ASSIST. As

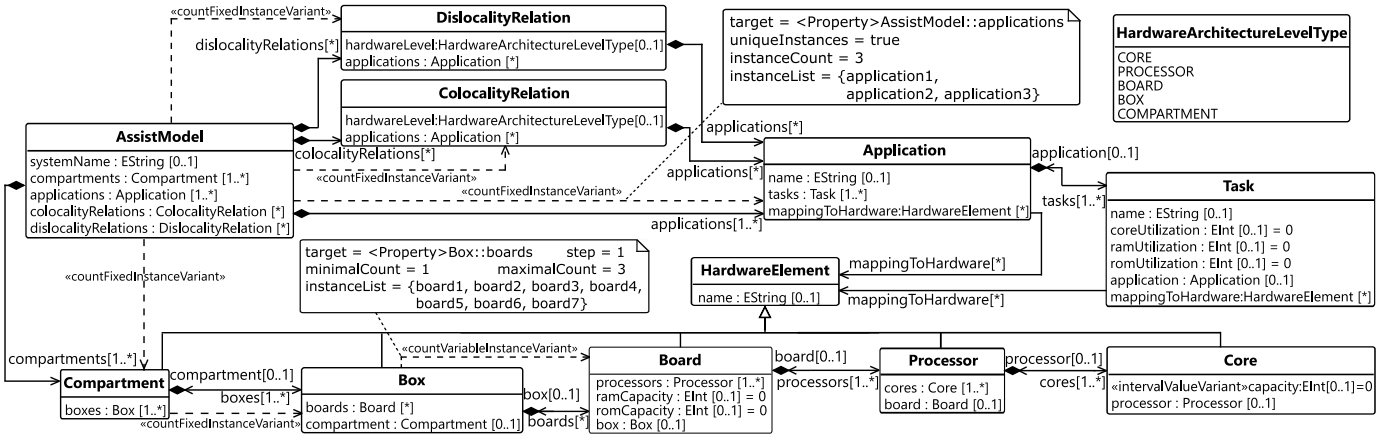

Fig. 3. UML model for mapping problems including architecture variant description

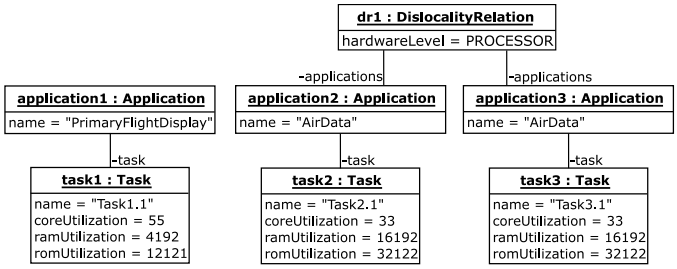

Fig. 4. Specification of existing applications and restrictions

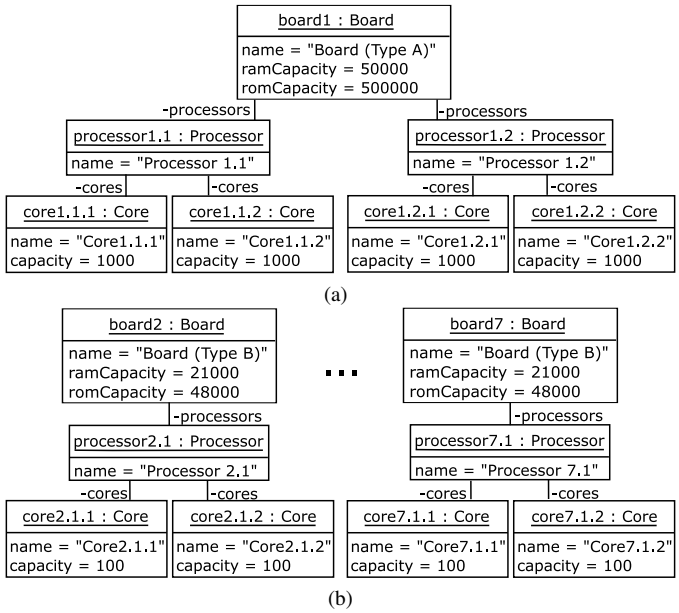

Fig. 5. Specification of board types including their processors and cores

a result, the hardware architecture is computed that contains the fewest elements while fulfilling all software process constraints, and with the smallest amount of unused resources.

Theoretically, 120 valid architectures can be designed based on the variant specification. It should be noted that this only counts the architecture selection problem, not the task

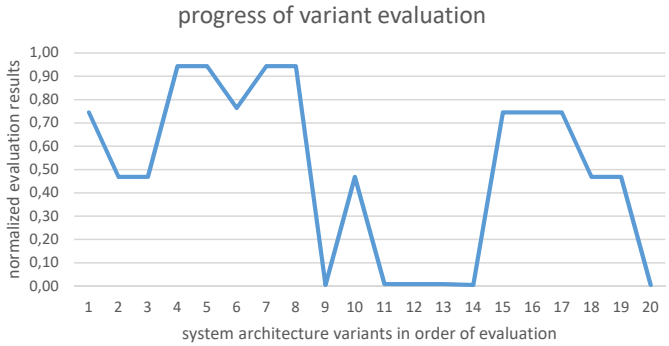

Fig. 6. Intermediate values of cost function during optimization

mapping aspect that is nested within. The system architecture optimization tool actually evaluated 20 different architectures during the simulated annealing process. An architecture with the best cost function is found after the ninth iteration. It is characterized by the use of two mainframe boards and the assignment of tasks to different cores. In this simple demonstration case the theoretical optimum was found, which can be determined manually for the case study. The simulated annealing procedure in our case continued for several more iterations before convergence, which later find other similar architectures which also have the optimal cost function.

VI. Conclusion

The formal optimization problem underlying the engineering task of design choices for a complex system is the basis for any (semi-)automatic methodology or tool support in early design stages. The paper overcomes the usual assumption of an unstructured set of design parameters by analyzing types of design parameters and their evolution in top-down hierarchical design approaches, where a structural architecture decision may lead to new design parameters on a lower level of detail. The formal concept of *existence-dependent parameters* is introduced with this regard, and some properties of the relations between the design parameters are derived. This allows to describe the corresponding design space as a

nested problem, for which adapted optimization methods are advantageous that exploit the locality of dependent parameters.

As an engineering proof of this idea the paper introduces a fully model-based integrated methodology for such problems together with a prototype tool that combines existing domain-specific methods. Approach and tool are validated with an application example, namely a distributed embedded systems design including hardware architecture and software mapping optimization.

In the future we will extend the method by adapted optimization heuristics to exploit the dependency structure of parameters.

ACKNOWLEDGMENT

The authors acknowledge the financial support for this work by the Federal Ministry of Education and Research of Germany (BMBF) in the projects "ARAMiS II" (DLR, grant identifier 01IS16025D) and "ARKOSE" (TU Ilmenau, grant identifier 01IS13031B).

REFERENCES

[1] B. S. Blanchard and W. J. Fabrycky, *Systems Engineering and Analysis*, 5th ed., ser. International Series in Industrial & Systems Engineering. Prentice Hall, 2010.

[2] B. P. Douglass, *Agile Systems Engineering*. Morgan Kaufmann, 2015.

[3] D. D. Walden, G. J. Roedler, K. J. Forsberg, R. D. Hamelin, and T. M. Shortell, Eds., *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities*, 4th ed. Wiley, 2015.

[4] R. Haberfellner, O. L. de Weck, E. Fricke, and S. Vössner, *Systems Engineering: Grundlagen und Anwendung*, 13th ed. Orell Füssli, 2015.

[5] A. L. Ramos, J. V. Ferreira, and J. Barceló, "Model-based systems engineering: An emerging approach for modern systems," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 42, no. 1, pp. 101–111, Jan. 2012.

[6] S. Jäger, R. Maschotta, T. Jungebloud, A. Wichmann, and A. Zimmermann, "Model-driven development of simulation-based system design tools," in *14th IEEE/ACIS Int. Conf. on Software Engineering Research, Management and Applications (SERA2016)*, Towson/Baltimore, MD, USA, Jun. 2016, pp. 209–215.

[7] A. Wichmann, F. Bedini, R. Maschotta, and A. Zimmermann, "Deriving architecture design variants for system optimization from design space descriptions expressed using a UML profile," in *Int. Symposium on Model-driven Approaches for Simulation Engineering (Mod4Sim 2017), part of SpringSim 2017*, Virginia Beach, VA (USA), Apr. 2017, pp. 707–718.

[8] A. Wichmann, R. Maschotta, F. Bedini, S. Jäger, and A. Zimmermann, "A UML profile for the specification of system architecture variants supporting design space exploration and optimization," in *Proc. 5th Int. Conf on Model-Driven Engineering and Software Development (MODELSWARD 2017)*, Porto, Portugal, Feb. 2017, pp. 418–426.

[9] J. Keinert, M. Streubühr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith, "SystemCoDesigner — an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 14, no. 1, pp. 1:1–1:23, Jan. 2009.

[10] E. Zitzler, L. Thiele, M. Laumanns, C. Fonseca, and V. da Fonseca, "Performance assessment of multiobjective optimizers: an analysis and review," *Evolutionary Computation, IEEE Transactions on*, vol. 7, no. 2, pp. 117 – 132, Apr. 2003.

[11] R. Dureja and K. Y. Rozier, "FuseIC3: An algorithm for checking large design spaces," in *2017 Formal Methods in Computer Aided Design (FMCAD)*, Oct. 2017, pp. 164–171.

[12] H. Zhu, "Approximation effects and user-controllable design space exploration," in *2017 Annual IEEE Int. Systems Conference (SysCon)*, Apr. 2017, pp. 1–8.

[13] S. Künzli, "Efficient design space exploration for embedded systems," PhD thesis, ETH Zurich, Apr. 2006.

[14] J. Eder, S. Zverlov, S. Voss, M. Khalil, and A. Ipatiov, "Bringing DSE to life: Exploring the design space of an industrial automotive use case," in *2017 ACM/IEEE 20th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS)*, Sep. 2017, pp. 270–280.

[15] S. Venkataramani, J. Choi, V. Srinivasan, K. Gopalakrishnan, and L. Chang, "POSTER: Design space exploration for performance optimization of deep neural networks on shared memory accelerators," in *2017 26th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2017, pp. 146–147.

[16] P. van Stralen and A. Pimentel, "Fast scenario-based design space exploration using feature selection," in *ARCS 2012*, Feb. 2012, pp. 1–7.

[17] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna, "UML-based multiprocessor SoC design framework," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 281–320, May 2006.

[18] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, "Daedalus: Toward composable multimedia MP-SoC design," in *Proceedings of the 45th Annual Design Automation Conference*, ser. DAC '08. New York, NY, USA: ACM, 2008, pp. 574–579.

[19] A. D. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[20] M. Belwal and T. S. B. Sudarshan, "A survey on design space exploration for heterogeneous multi-core," in *2014 International Conference on Embedded Systems (ICES)*, July 2014, pp. 80–85.

[21] W. Damm, A. Metzner, F. Eisenbrand, G. Shmonin, R. Wilhelm, and S. Winkel, "Mapping Task-Graphs on Distributed ECU Networks: Efficient Algorithms for Feasibility and Optimality," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, 2006, pp. 87–90.

[22] S. Kugele, W. Haberl, M. Tautschnig, and M. Wechs, "Optimizing automatic deployment using non-functional requirement annotations," in *Leveraging Applications of Formal Methods, Verification and Validation*, ser. Comm. in Computer and Information Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2009, vol. 17, pp. 400–414.

[23] A. Zimmermann, *Stochastic Discrete Event Systems — Modeling, Evaluation, Applications*. Berlin Heidelberg New York: Springer, 2007.

[24] A. Wichmann, S. Jäger, T. Jungebloud, R. Maschotta, and A. Zimmermann, "System architecture optimization with runtime reconfiguration of simulation models," in *IEEE Int. Systems Conference (IEEE SysCon 2015)*, Vancouver, Canada, Apr. 2015.

[25] A. Wichmann, S. Jäger, T. Jungebloud, R. Maschotta, and A. Zimmermann, "Specification and execution of system optimization processes with UML activity diagrams," in *Proc. IEEE Systems Conference (SysCon 2016)*, 2016, pp. 1–7.

[26] T. Givargis, F. Vahid, and J. Henkel, "System-level exploration for Pareto-optimal configurations in parameterized system-on-a-chip," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 4, pp. 416–422, Aug 2002.

[27] A. D. Pimentel, "Exploring exploration: A tutorial introduction to embedded systems design space exploration," *IEEE Design Test*, vol. 34, no. 1, pp. 77–90, Feb. 2017.

[28] R. Hilbrich and M. Behrisch, "Experiences gained from modeling and solving large mapping problems during system design," in *IEEE Systems Conference 2017*, Feb. 2017, pp. 620–627.

[29] R. Hilbrich, "Architecture synthesis for safety critical systems — ASSIST," online, 2014. [Online]. Available: https://github.com/RobertHilbrich

[30] C. Prud'homme, J.-G. Fages, and X. Lorca, *Choco Documentation*, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016. [Online]. Available: http://www.choco-solver.org

[31] R. Hilbrich and L. Dieudonne, "Deploying safety-critical applications on complex avionics hardware architectures," *Journal of Software Engineering and Applications*, vol. 06, no. 05, pp. 229–235, 2013.

[32] M. Große-Rhode, R. Hilbrich, S. Mann, and S. Weißleder, *System Quality and Software Architecture*, 1st ed. Morgan Kaufmann Publishers, 2014, ch. Chapter 9: Achieving Quality in Customer-Configurable Products, pp. 233–261.

[33] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to Integrated Modular Avionics," in *Digital Avionics Systems Conference, 2007. DASC '07. IEEE/AIAA 26th*, 2007, pp. 2.A.1–1–2.A.1–10.